**Entity Framework Core**

**Se a aplicação não sofresse mudanças, não seria necessário usarmos o Entity. Se o cliente solicitar que seja removido o campo categoria da classe Produto, teríamos que tratar todas as SQL's, caso contrário não conseguiríamos compilar o código.**

**A aplicação ainda é pequena e já tivemos um grande trabalho. Imagine um aplicação enorme com várias classes de negócio, o impacto seria enorme, o que dificultaria a evolução da aplicação.**

**O ideal seria que alguém ficasse responsável pelo trabalho de criar as SQL's, passar parâmetros e assim por diante. É aí que entra o Entity Framework.**

## **Instalação do Visual Studio e SQL Server**

**Como pré-requisito de aplicações, você vai precisar do Visual Studio 2017 Community e do SQL Server Express. Para baixar os dois, é só visitar a** [**página do Visual Studio**](https://www.visualstudio.com/pt-br/vs) **e obter o instalador do Visual Studio 2017. O SQL Server já vem embutido nele.**

## **Download do projeto**

**Para rodar os exemplos dos vídeos em sua máquina, baixe o projeto inicial no link** [**aqui**](https://s3.amazonaws.com/caelum-online-public/635-entity-framework-core/01/projetos/Alura.Loja.zip)**.**

**Depois que você fizer o download do projeto, basta descompactá-lo numa pasta qualquer para abrí-lo no Visual Studio. Aqui colocamos o projeto na pasta que o VS usa como padrão para guardar seus projetos (no Windows 10, fica em c:\users\<nome do seu usuário>\Documents\Visual Studio 2017\Projects).**

**Já no Visual Studio, vamos abrir a solução clicando em Arquivo, Abrir, Projeto\Solução, e escolhendo a pasta que descompactamos no passo anterior. A solução é do tipo Console Application, apenas para mostrar os conceitos e ferramentas que serão discutidas no curso.**

## **Criação do banco de dados**

**Lembrando que esse curso é uma introdução ao Entity Framework Core e irá aplicar sua teoria em um banco de dados completamente novo. Se sua necessidade é aprender como integrar o EF Core a um banco de dados já existente, temos um outro curso dividido em duas partes voltado para esse propósito:**

* **[Entity Framework Core parte 1: Mapeando um banco pré-existente](https://cursos.alura.com.br/course/entity-framework-core-banco-pre-existente-parte1)**
* **[Entity Framework Core parte 2: Mapeando um banco pré-existente](https://cursos.alura.com.br/course/entity-framework-core-banco-pre-existente-parte2)**

**Outro passo necessário para rodar os exemplos é criar o banco de dados utilizado nessa solução. Abra o Pesquisador de Objetos do SQL Server no menu Exibir e clique na aba SQL Server para mostrar os servidores disponíveis em sua máquina. Abra o servidor (localdb), clique com o botão direito em cima da aba *Bancos de Dados* e escolha a opção *Adicionar Novo Banco de Dados*.**

**Na janela de diálogo a seguir, defina como nome do banco LojaDB e aceite a sugestão do VS para a localização. Ao clicar em OK será criado o banco.**

## **Criação da tabela Produtos**

**Agora que temos um banco, vamos criar a tabela de produtos. Primeiro, abra o arquivo ddl-produtos.txt, que está disponível no projeto que você baixou, e copie todo o seu conteúdo usando Ctrl+C. Abra o banco recém criado *LojaDB* e com o botão direito na aba *Tabelas*, escolha a opção *Adicionar Nova Tabela...*. Na janela que será aberta, procure a aba *T-SQL* e substitua todo o código nela pelo texto que você copiou na área de transferência. Clique no botão *Atualizar* e em seguida *Atualizar Banco de Dados*. Confira que a tabela *Produtos* está criada no banco de dados *LojaDB*.**

**Já instalamos o Entity, vamos começar a usá-lo. Comentaremos a linha GravarUsandoAdoNet(); que está dentro do método Main() da classe Program. Em seguida, colocaremos uma nova chamada para o método GravarUsandoEntity(), que usamos para persistir os dados com o Entity.**

**static void Main(string [] args)**

**{**

**//GravarUsandoAdoNet();**

**GravarUsandoEntity();**

**}**

**Criaremos o método GravarUsandoEntity(), e dentro dele colocaremos todo o conteúdo contido no método GravarUsandoAdoNet().**

**static void Main(string [] args)**

**{**

**//GravarUsandoAdoNet();**

**GravarUsandoEntity();**

**}**

**private static void GravarUsandoEntity()**

**{**

**Produto p = new Produto();**

**p.Nome = "Harry Potter e a Ordem da Fênix";**

**p.Categoria = "Livros";**

**p.Preco = 19.89;**

**using (var repo = new ProdutoDAO())**

**{**

**repo.Adicionar(p);**

**}**

**}**

**Vamos fazer algumas modificações no método GravarUsandoEntity(). A estratégia do ADO.Net, era criar uma classe que representasse o *Data Access Object* de um modelo que seria persistido.**

**O Entity trabalha um pouco diferente, criaremos uma classe que vai representar e permitir persistirmos todas as classes necessárias, e não apenas uma específica como a de Produto.**

**Essa classe possui o conceito no Entity de contexto da aplicação. Como o modelo de negócios do projeto é uma loja, então mudaremos a chamada ao DAO para a classe LojaContext(). É uma convenção chamarmos as classe pelo modelo de negócios seguido do sufixo *Context*. Também trocaremos o nome da variável repo para contexto.**

**private static void GravarUsandoEntity()**

**{**

**Produto p = new Produto();**

**p.Nome = "Harry Potter e a Ordem da Fênix";**

**p.Categoria = "Livros";**

**p.Preco = 19.89;**

**using (var contexto = new LojaContext())**

**{**

**contexto.Adicionar(p);**

**}**

**}**

**O Visual Studio mostra um erro após a alteração, isso porque não criamos a classe ainda. Vamos criá-la. A classe criada ficará da seguinte maneira:**

**using System;**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**public class LojaContext : IDisposable**

**{**

**}**

**}**

**Primeiramente faremos com que a classe permita *usar a API do Entity* dentro dela. Para que isso aconteça, faremos a classe *herdar* de DbContext.**

**using Microsoft.EntityFrameworkCore;**

**using System;**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**public class LojaContext : DbContext**

**{**

**}**

**}**

**Informaremos *quais classe serão persistidas pelo Entity. Mas como será feito? Faremos isso por meio de uma propriedade que vai representar o conjunto de objetos da classe Produto. A propriedade ficará definida com o tipo DbSet<Produto>, e o nome da propriedade colocaremos o \*mesmo nome* da tabela do banco de dados.**

**using Microsoft.EntityFrameworkCore;**

**using System;**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**public class LojaContext : DbContext**

**{**

**public DbSet<Produto> Produtos { get; set; }**

**}**

**}**

**Ainda na classe, definiremos qual é o *banco de dados* e qual é o *endereço*. Para isso, iremos sobrescrever o método OnConfiguring() da classe DbContext.**

**using Microsoft.EntityFrameworkCore;**

**using System;**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**public class LojaContext : DbContext**

**{**

**public DbSet<Produto> Produtos { get; set; }**

**protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)**

**{**

**}**

**}**

**}**

**Esse método tem como argumento de entrada, um *construtor de opções*. Usando o optionsBuilder chamaremos o método UseSqlServer(), passando como argumento o endereço do banco que usamos no ProdutoDAO.**

**using Microsoft.EntityFrameworkCore;**

**using System;**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**public class LojaContext : DbContext**

**{**

**public DbSet<Produto> Produtos { get; set; }**

**protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)**

**{**

**optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=LojaDB;Trusted\_Connection=true;")**

**}**

**}**

**}**

**Na classe Program dentro do using, substituiremos a chamada repo.Adicionar(p) por um código que usará o Entity. Tudo o que quisermos fazer no banco de dados, vai passar em uma instância do LojaContext.**

**Usando a variável contexto, chamaremos a propriedade Produtos que por sua vez chamará o método Add(), que recebe o objeto que iremos persistir.**

**private static void GravarUsandoEntity()**

**{**

**Produto p = new Produto();**

**p.Nome = "Harry Potter e a Ordem da Fênix";**

**p.Categoria = "Livros";**

**p.Preco = 19.89;**

**using (var contexto = new LojaContext())**

**{**

**contexto.Produtos.Add(p);**

**}**

**}**

**Em seguida, basta salvar as mudanças usando o método contexto.SaveChanges().**

**using (var contexto = new LojaContext())**

**{**

**contexto.Produtos.Add(p);**

**contexto.SaveChanges();**

**}**

**Isso é tudo o que precisamos para salvar um produto no banco de dados usando o Entity. Rodaremos a aplicação com o "Ctrl + F5". Se olharmos no banco de dados, veremos que o produto foi salvo.**

**Você deve estar se perguntando se o Entity suporta transações. Por padrão, desde a versão 6, quando o Entity executa o método SaveChanges uma transação é criada para englobar os comandos SQL necessários para salvar as alterações do contexto no banco.**

**Se você precisar de um maior controle sobre transações em sua aplicação, poderá usar os métodos BeginTransaction e UseTransaction, disponíveis na propriedade Database da classe DbContext.**

**Para maiores informações sobre o assunto, leia a** [**página**](https://msdn.microsoft.com/en-us/library/dn456843.aspx) **(em inglês) no site da Microsoft.**

**Suponha que sua aplicação deverá suportar o MySQL. Atualmente o código de LojaContext está acoplado diretamente ao SQL Server. Como podemos tornar LojaContext independente do banco de dados empregado na aplicação?**

**Usando o princípio da Injeção de Dependência! Criaremos um construtor que receberá um objeto do tipo DbContextOptions<LojaContext> com as opções necessárias para usar o MySQL. Ele chamará o construtor da classe base DbContext através da palavra reservada base, passando as opções como argumento. Para que o código continue compilando, manteremos o construtor sem argumentos.**

**Quando tanto o construtor como o método OnConfiguring são usados, o último sobrescreve as opções no contexto. Por isso precisamos colocar uma condição no método OnConfiguring perguntando se as opções já foram configuradas, o que é obtido através da verificação da propriedade booleana IsConfigured.**

**O código final fica assim:**

**using Microsoft.EntityFrameworkCore;**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**public class LojaContext : DbContext**

**{**

**public DbSet<Produto> Produtos { get; set; }**

**public LojaContext()**

**{ }**

**public LojaContext(DbContextOptions<LojaContext> options): base(options)**

**{ }**

**protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)**

**{**

**if (!optionsBuilder.IsConfigured)**

**{**

**optionsBuilder**

**.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=LojaDB;Trusted\_Connection=true;");**

**}**

**}**

**}**

**}Repare que é possível incluir mais de um produto antes de enviar as mudanças para o banco de dados.**

**Produto p1 = new Produto();**

**p1.Nome = "Harry Potter e a Ordem da Fênix";**

**p1.Categoria = "Livros";**

**p1.Preco = 19.89;**

**Produto p2 = new Produto();**

**p2.Nome = "Senhor dos Anéis 1";**

**p2.Categoria = "Livros";**

**p2.Preco = 19.89;**

**Produto p3 = new Produto();**

**p3.Nome = "O Monge e o Executivo";**

**p3.Categoria = "Livros";**

**p3.Preco = 19.89;**

**contexto.Produtos.Add(p1);**

**contexto.Produtos.Add(p2);**

**contexto.Produtos.Add(p3);**

**contexto.SaveChanges();**

**Melhor ainda, como vimos em um exercício anterior, essas mudanças farão parte da mesma transação. Mesmo assim a classe DbSet possui outro método para incluir objetos no contexto. É o método AddRange.**

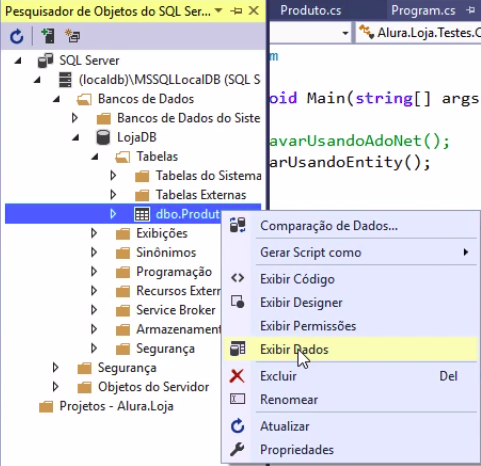
**contexto.Produtos.AddRange(p1, p2, p3);**

**contexto.SaveChanges();**

**Qual a diferença entre os métodos Add e AddRange, além de um aceitar como argumento um objeto e outro uma lista? O segundo pode trazer uma melhora significativa de performance para inclusão de muitos objetos.**

**Na aula anterior, conhecemos o Entity e seus benefícios, instalamos no projeto e inserimos um produto no banco de dados usando o Entity. Nessa aula, vamos explorar as operações do *CRUD* com a classe Produto.**

**Para consultar o produto inserido no banco, é necessário acessar o "Pesquisador de Objetos do SQL Server", clicar com o botão direito na tabela e selecionar "Exibir Dados".**

****

**Faremos essa operação de consultar os dados do banco, mas pelo C#. Na classe ProdutoDAO, já temos o método Produtos() que faz essa operação, mas usando o ADO.Net.**

**O que basicamente o método Produtos() faz, é montar e executar o comando SQL select. Em seguida, executa um laço para percorrer todo o resultado e convertê-lo em um lista de produtos.**

**Mas como faremos usando o Entity? Na classe Program dentro do método Main(), faremos uma chamada para o método RecuperaProdutos() que ainda não foi criado. Deixaremos comentado a chamada do método GravarUsandoEntity(), para que não seja adicionado um novo produto.**

**static void Main(string[] args)**

**{**

**//GravarUsandoAdoNet();**

**//GravarUsandoEntity();**

**RecuperarProdutos();**

**}**

**Criaremos o método RecuperaProdutos(). A estratégia será a mesma usada para adicionar o produto, criaremos uma instância da classe LojaContext que representa o contexto do Entity na aplicação.**

**static void Main(string[] args)**

**{**

**//GravarUsandoAdoNet();**

**//GravarUsandoEntity();**

**RecuperarProdutos();**

**}**

**private static void RecuperaProdutos()**

**{**

**using(var repo = new LojaContext())**

**{**

**}**

**}**

**Com o objeto repo, pediremos para que ele nos retorne uma lista de produtos chamando repo.Produto.ToList(). Receberemos como resultado uma lista de produtos, que guardaremos uma variável chamada produtos.**

**private static void RecuperaProdutos()**

**{**

**using (var repo = new LojaContext())**

**{**

**IList<Produto> produtos = repo.Produtos.ToList();**

**}**

**}**

**Para conferirmos o resultado, usaremos o foreach com a lista produtos, imprimindo o nome de cada um dos elementos.**

**private static void RecuperaProdutos()**

**{**

**using (var repo = new LojaContext())**

**{**

**IList<Produto> produtos = repo.Produtos.ToList();**

**foreach (var item in produtos)**

**{**

**Console.WriteLine(item.Nome);**

**}**

**}**

**}**

**Rodando a aplicação com "Ctrl + F5"**

**contexto.Clientes.ToList();**

**Isso aí. Através da propriedade Clientes acessamos os métodos de DbSet. Um deles é ToList, para recuperar a lista de todos os clientes no banco.**

**Já temos o método Remover() na classe ProdutoDAO que faz o processo de remoção dos dados persistidos no banco. O método basicamente monta e executa o comando SQL delete, passando o Id do produto.**

**Na classe Program dentro do método Main(), vamos fazer uma chamada para o método ExcluirProdutos(). Em seguida implementaremos o método, pegando as instância da classe LojaContext.**

**static void Main(string[] args)**

**{**

**//GravarUsandoAdoNet();**

**//GravarUsandoEntity();**

**RecuperarProdutos();**

**ExcluirProdutos();**

**}**

**private static void ExcluirProdutos()**

**{**

**using(var repo = new LojaContext())**

**{**

**}**

**}**

**Para remover os produtos, antes é preciso pegá-los, visto que não podemos remover um produto que não sabemos. Pegaremos os produtos da mesma forma que fizemos no método RecuperarProdutos().**

**static void Main(string[] args)**

**{**

**//GravarUsandoAdoNet();**

**//GravarUsandoEntity();**

**RecuperarProdutos();**

**ExcluirProdutos();**

**}**

**private static void ExcluirProdutos()**

**{**

**using(var repo = new LojaContext())**

**{**

**IList<Produto> produtos = repo.Produtos.ToList();**

**}**

**}**

**Com a lista produtos em mãos, vamos percorrê-la. Dessa forma, passaremos cada item como argumento ao método Remover() da propriedade repo.Produto.**

**static void Main(string[] args)**

**{**

**//GravarUsandoAdoNet();**

**//GravarUsandoEntity();**

**RecuperarProdutos();**

**ExcluirProdutos();**

**}**

**private static void ExcluirProdutos()**

**{**

**using(var repo = new LojaContext())**

**{**

**IList<Produto> produtos = repo.Produtos.ToList();**

**foreach(var item in produtos)**

**{**

**repo.Produtos.Remove(item);**

**}**

**}**

**}**

**Após o encerramento do laço, pediremos para o repositório salvar as alterações feitas.**

**static void Main(string[] args)**

**{**

**//GravarUsandoAdoNet();**

**//GravarUsandoEntity();**

**RecuperarProdutos();**

**ExcluirProdutos();**

**}**

**private static void ExcluirProdutos()**

**{**

**using(var repo = new LojaContext())**

**{**

**IList<Produto> produtos = repo.Produtos.ToList();**

**foreach(var item in produtos)**

**{**

**repo.Produtos.Remove(item);**

**}**

**repo.SaveChanges();**

**}**

**}**

**Para testar a remoção sem que seja necessário acessar o banco de dados e verificar a remoção do produto, dentro do método Main() após a chamada ao ExcluirProdutos(), faremos outra chamada ao RecuperarProdutos().**

**static void Main(string[] args)**

**{**

**//GravarUsandoAdoNet();**

**//GravarUsandoEntity();**

**RecuperarProdutos();**

**ExcluirProdutos();**

**RecuperarProdutos();**

**}**

**private static void ExcluirProdutos()**

**{**

**using(var repo = new LojaContext())**

**{**

**IList<Produto> produtos = repo.Produtos.ToList();**

**foreach(var item in produtos)**

**{**

**repo.Produtos.Remove(item);**

**}**

**repo.SaveChanges();**

**}**

**}**

**Como estamos removendo todos os produtos do banco de dados, o laço do método RecuperaProdutos() não será executado, consequentemente não executando o Console.WriteLine(item.Nome), o que não nos indicaria que a remoção foi efetuada com sucesso. Para visualizarmos melhor o teste, vamos adicionar um Console.WriteLine() antes do laço.**

**static void Main(string[] args)**

**{**

**//GravarUsandoAdoNet();**

**//GravarUsandoEntity();**

**RecuperarProdutos();**

**ExcluirProdutos();**

**RecuperarProdutos();**

**}**

**private static void ExcluirProdutos()**

**{**

**using(var repo = new LojaContext())**

**{**

**IList<Produto> produtos = repo.Produtos.ToList();**

**foreach(var item in produtos)**

**{**

**repo.Produtos.Remove(item);**

**}**

**repo.SaveChanges();**

**}**

**}**

**private static void RecuperarProdutos()**

**{**

**using (var repo = new LojaContext())**

**{**

**IList<Produto> produtos = repo.Produtos.ToList();**

**Console.WriteLine("foram encontrados {0} produto(s)", produtos.Count);**

**foreach(var item in produtos)**

**{**

**Console.WriteLine(item.Nome);**

**}**

**}**

**}**

**Executando a aplicação, vemos que tudo funcionou como o esperado.**

**Aprenderemos a atualizar um produto. Na classe ProdutoDAO, temos o método Atualizar(). O método recebe um produto, monta o comando SQL, criar e coloca os parâmetro no comando e por fim, executa. Agora, utilizaremos o Entity.**

**Dentro do método Main() da classe Program, comentaremos todas as chamadas de métodos, e em seguida, faremos uma nova chamada ao método AtualizarProduto() que ainda não existe.**

**static void Main(string[] args)**

**{**

**//GravarUsandoAdoNet();**

**//GravarUsandoEntity();**

**//RecuperarProdutos();**

**//ExcluirProdutos();**

**//RecuperarProdutos();**

**AtualizarProduto();**

**}**

**Após isso, criaremos o método AtualizarProduto(). Porém, para verificarmos a atualização do produto, antes é necessário incluir um produto - já que removemos todos do banco de dados no vídeo anterior - e mostrá-lo, efetuar a atualização e mostrá-lo novamente.**

**static void Main(string[] args)**

**{**

**//GravarUsandoAdoNet();**

**//GravarUsandoEntity();**

**//RecuperarProdutos();**

**//ExcluirProdutos();**

**//RecuperarProdutos();**

**AtualizarProduto();**

**}**

**private static void AtualizarProduto()**

**{**

**// inclui um produto**

**RecuperarProdutos();**

**// atualiza o produto**

**RecuperarProdutos();**

**}**

**O primeiro passo é simples, basta chamarmos o método GravarUsandoEntity()para que seja incluído no banco de dados o produto já preparado no método.**

**private static void AtualizarProduto()**

**{**

**// inclui um produto**

**GravarUsandoEntity();**

**RecuperarProdutos();**

**// atualiza o produto**

**RecuperarProdutos();**

**}**

**Com o produto incluído, vamos criar um objeto que é a instância da classe LojaContext.**

**private static void AtualizarProduto()**

**{**

**// inclui um produto**

**GravarUsandoEntity();**

**RecuperarProdutos();**

**// atualiza o produto**

**using(var repo = LojaContext())**

**{**

**}**

**RecuperarProdutos();**

**}**

**Com o objeto, em vez de pedirmos uma lista de produtos, vamos pedir apenas o primeiro encontrado e armazená-lo em um variável chamada primeiro.**

**private static void AtualizarProduto()**

**{**

**// inclui um produto**

**GravarUsandoEntity();**

**RecuperarProdutos();**

**// atualiza o produto**

**using(var repo = LojaContext())**

**{**

**Produto primeiro = repo.Produtos.First();**

**}**

**RecuperarProdutos();**

**}**

**O First() irá retornar o primeiro produto encontrado no banco de dados. Basta agora modificarmos o nome do produto e passá-lo como argumento para o repo.Produto.Update(). Sem esquecer de no final salvar as alterações.**

**static void Main(string[] args)**

**{**

**//GravarUsandoAdoNet();**

**//GravarUsandoEntity();**

**//RecuperarProdutos();**

**//ExcluirProdutos();**

**//RecuperarProdutos();**

**AtualizarProduto();**

**}**

**private static void AtualizarProduto()**

**{**

**// inclui um produto**

**GravarUsandoEntity();**

**RecuperarProdutos();**

**// atualiza o produto**

**using (var repo = new LojaContext())**

**{**

**Produto primeiro = repo.Produtos.First();**

**primeiro.Nome = "Cassino Royale - Editado";**

**repo.Produtos.Update(primeiro);**

**repo.SaveChanges();**

**}**

**RecuperarProdutos();**

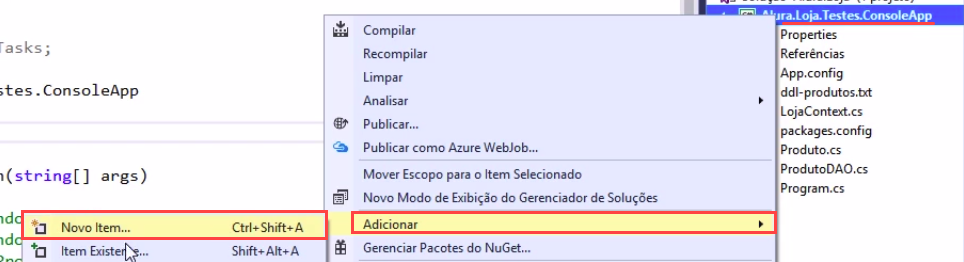
**}**

**Executaremos a classe e veremos que tudo ocorreu como esperado. Porém, a nossa classe com os métodos que usam o Entity foi montada apenas para testes. No próximo vídeo veremos como ficaria uma classe *DAO* usando o Entity.**

**Aprendemos a trabalhar com todas as operações do *CRUD*, porém fizemos isso em uma classe de teste. Criaremos uma nova classe ProdutoDAO, mas dessa vez fazendo acesso aos dados usando o Entity.**

**Antes de implementarmos a ProdutoDAO, vamos utilizar um recurso de Orientação a Objetos que vai nos auxiliar na criação da classe. Como as classes *DAO*, tanto a que usa Entity e a que usa ADO.Net, possuem os mesmo métodos, podemos afirmar que elas assinam o mesmo contrato. Partindo dessa ideia, criaremos a interface IProdutoDAO.**

**Clicando com o botão direito em Alura.Loja.Teste.ConsoleApp, selecionaremos a opção "Adicionar > Novo Item".**

****

**Escolheremos a opção "Interface", colocando o nome IProdutoDAO. A interface irá conter as quatro operações do *CRUD*, adicionar, atualizar, remover, e listar.**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**interface IProdutoDAO**

**{**

**void Adicionar(Produto p);**

**void Atualizar(Produto p);**

**void Remover(Produto p);**

**IList<Produto> Produtos();**

**}**

**}**

**Faremos a classe ProdutoDAO que utiliza o ADO.Net implementar a interface. Além disso, é necessário mudar a visibilidade internal dos métodos para public.**

**internal class ProdutoDAO : IDisposable, IProdutoDAO**

**{**

**//...**

**public void Adicionar(Produto p)**

**{**

**//...**

**}**

**public void Atualizar(Produto p)**

**{**

**//...**

**}**

**public void Remover(Produto p)**

**{**

**//...**

**}**

**public IList<Produto> Produtos()**

**{**

**//...**

**}**

**}**

**Da mesma forma como criamos a interface, criaremos um nova classe chamada ProdutoDAOEntity. Faremos a nova classe implementar a interface IDisposablee IProdutoDAO.**

**class ProdutoDAOEntity : IProdutoDAO, IDisposable**

**{**

**public void Adicionar(Produto p)**

**{**

**throw new NotImplementedException();**

**}**

**public void Atualizar(Produto p)**

**{**

**throw new NotImplementedException();**

**}**

**public void Dispose()**

**{**

**throw new NotImplementedException();**

**}**

**public IList<Produto> Produtos()**

**{**

**throw new NotImplementedException();**

**}**

**public void Remover(Produto p)**

**{**

**throw new NotImplementedException();**

**}**

**}**

**Agora na classe Program, em vez de criarmos os objetos da classe LojaContext, criaremos de ProdutoDAOEntity. Dessa forma perderemos todos os métodos relacionados ao LojaContext, sendo necessário trocar pela chamada ao respectivo método de ProdutoDAOEntity.**

**static void Main(string[] args)**

**{**

**//Chamada de Métodos...**

**}**

**private static void AtualizaProduto()**

**{**

**// inclui um produto**

**// atualiza o produto**

**using(var repo = new ProdutoDAOEntity())**

**{**

**Produto primeiro = repo.Produtos().First();**

**primeiro.Nome = "Cassino Royale - Editado";**

**repo.Atualizar(primeiro);**

**}**

**RecuperaProdutos();**

**}**

**private static void ExcluirProdutos()**

**{**

**using (var repo = new ProdutoDAOEntity())**

**{**

**IList<Produto> produtos = repo.Produtos();**

**foreach (var item in produtos)**

**{**

**repo.Remover(item);**

**}**

**}**

**}**

**private static void RecuperarProdutos()**

**{**

**using (var repo = new ProdutoDAOEntity())**

**{**

**IList<Produto> produtos = repo.Produtos();**

**Console.WriteLine("Foram encontrados {0} produto(s).", produtos.Count);**

**foreach (var item in produtos)**

**{**

**Console.WriteLine(item.Nome);**

**}**

**}**

**}**

**private static void GravarUsandoEntity()**

**{**

**Produto p = new Produto();**

**p.Nome = "Cassino Royale";**

**p.Categoria = "Filmes";**

**p.Preco = 19.89;**

**using(var repo = new ProdutoDAOEntity())**

**{**

**repo.Adicionar(p);**

**}**

**}**

**Fizemos a troca, tiramos de LojaContext e passamos a usar ProdutoDAOEntity. Entretanto, a classe ProdutoDAOEntity ainda não está usando o Entity. Para utilizarmos qualquer método Entity, precisamos acessar a partir de um objeto de contexto na aplicação. Por isso, criaremos um *atributo* privado do tipo LojaContext.**

**class ProdutoDAOEntity : IProdutoDAO, IDisposable**

**{**

**private LojaContext contexto;**

**//...**

**}**

**Para incluir um produto, dentro do método Adicionar() vamos apagar todo o conteúdo e invocar o método Add() da propriedade contexto.Produto. É necessário salvar as mudanças do contexto.**

**class ProdutoDAOEntity : IProdutoDAO, IDisposable**

**{**

**private LojaContext contexto;**

**public void Adicionar(Produto p)**

**{**

**contexto.Produtos.Add(p);**

**contexto.SaveChanges();**

**}**

**// ...**

**}**

**Para atualizar, faremos da mesma forma que fizemos para adicionar um produto, mas chamaremos o método Update().**

**class ProdutoDAOEntity : IProdutoDAO, IDisposable**

**{**

**// ...**

**public void Atualizar(Produto p)**

**{**

**contexto.Produtos.Update(p);**

**contexto.SaveChanges();**

**}**

**// ...**

**}**

**Para remover, chamaremos o método Remove().**

**class ProdutoDAOEntity : IProdutoDAO, IDisposable**

**{**

**// ...**

**public void Remover(Produto p)**

**{**

**contexto.Produtos.Remove(p);**

**contexto.SaveChanges();**

**}**

**// ...**

**}**

**Para listar os produtos basta retornarmos o método ToList():**

**class ProdutoDAOEntity : IProdutoDAO, IDisposable**

**{**

**// ...**

**public IList<Produto> Produtos()**

**{**

**return contexto.Produtos.ToList();**

**}**

**}**

**Ainda falta criarmos uma instância de LojaContext. Faremos isso no construtor da classe. No método Dispose(), chamaremos o contexto.Dispose()**

**class ProdutoDAOEntity : IProdutoDAO, IDisposable**

**{**

**private LojaContext contexto;**

**public ProdutoDAOEntity()**

**{**

**this.contexto = new LojaContext();**

**}**

**public void Dispose()**

**{**

**contexto.Dispose();**

**}**

**//...**

**}**

**Dessa forma, temos uma classe *DAO* que faz os acessos ao banco de dados de uma forma muito mais enxuta e limpa, com praticamente metade de linhas de código de uma classe com ADO.Net.**

**Na aula anterior aprendemos como o Entity faz as operações de CRUD. Nessa aula veremos como esse framework gerencia os objetos que serão persistidos no banco de dados. Para começarmos, apagaremos todos os métodos da classe Program, mantendo apenas o método Main() vazio.**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**}**

**}**

**}**

**Usaremos o objeto de contexto do Entity, que é uma instância da classe LojaContext, nós armazenaremos em uma variável chamada contexto. Com o objeto contexto, usaremos a propriedade contexto.Produtos chamando o método ToList().**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**using (var contexto = new LojaContext())**

**{**

**var produtos = contexto.Produtos.ToList();**

**}**

**}**

**}**

**}**

**Como queremos mostrar tudo o que temos no banco de dados, faremos um laço iterando pela lista produtos.**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**using(var contexto = LojaContext())**

**{**

**var produtos = contexto.Produtos.ToList();**

**foreach (var p in produtos)**

**{**

**Console.WriteLine(p);**

**}**

**}**

**}**

**}**

**}**

**Usaremos o "Ctrl + F5" para executar a aplicação. Recebemos como resultado o nome da classe em vez do produto, isso ocorre por que quando chamamos o Console.WriteLine() para a uma variável de referência, é chamado o método ToString() dessa variável. Olhando na classe Produto, não temos o método ToString() implementado, por isso vamos sobrescrevê-lo para que seja apresentado o nome do produto.**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**internal class Produto**

**{**

**public int Id { get; internal set; }**

**public string Nome { get; internal set; }**

**public string Categoria { get; internal set; }**

**public double Preco { get; internal set; }**

**public override string ToString()**

**{**

**return "Produto: " + this.nome;**

**}**

**}**

**}**

**Executando a aplicação com "Ctrl + F5" novamente, veremos os nomes dos produtos. Imaginando que o primeiro produto armazenado no banco de dados não esteja com o nome correto, poderemos atualizá-lo.**

**Começaremos pegando o primeiro produto da lista produtos com o método First() e armazenaremos em uma variável chamada p1. Em seguida, alteraremos o valor do nome e salvaremos as alterações. Para visualizarmos a mudança, vamos repetir o código onde pegamos todos os produtos do banco de dados e iteramos a lista com um laço de repetição.**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**using(var contexto = LojaContext())**

**{**

**var produtos = contexto.Produtos.ToList();**

**foreach (var p in produtos)**

**{**

**Console.WriteLine(p);**

**}**

**var p1 = produtos.First();**

**p1.Nome = "Harry Potter";**

**contexto.SaveChages();**

**Console.WriteLine("=================");**

**produtos = contexto.Produtos.ToList();**

**foreach (var p in produtos)**

**{**

**Console.WriteLine(p);**

**}**

**}**

**}**

**}**

**}**

**O resultado é o esperado. Mas como o Entity sabe que ao mudar uma propriedade de um dado na lista, ele precisa fazer o *Update* no banco de dados? O que acontece é que a classe que representa o contexto herda de DbContext. A instância chamada *Change Tracker* é responsável por rastrear todas as mudanças feitas no contexto. É possível recuperar todas as entidade usando o objeto contexto da seguinte forma:**

**contexto.ChangeTracker.Entries();**

**Vamos iterar a lista retornada pelo Entries() e imprimir os seus elementos. Comentaremos o trecho de código onde alteramos e mostramos o nome do produto.**

**static void Main(string[] args)**

**{**

**using(var contexto = LojaContext())**

**{**

**var produtos = contexto.Produtos.ToList();**

**foreach (var p in produtos)**

**{**

**Console.WriteLine(p);**

**}**

**Console.WriteLine("=================");**

**foreach (var e in contexto.ChangeTracker.Entries())**

**{**

**Console.WriteLine(e);**

**}**

**//var p1 = produtos.First();**

**//p1.Nome = "Harry Potter";**

**//contexto.SaveChages();**

**//Console.WriteLine("=================");**

**//produtos = contexto.Produtos.ToList();**

**//foreach (var p in produtos)**

**//{**

**// Console.WriteLine(p);**

**//}**

**}**

**}**

**Executando a aplicação, vemos que foi apresentado o nome da classe EntityEntry. Essa classe possui uma propriedade chama State que registra o estado da entidade, e caso ela tenha sido alterada, o SaveChanges() terá que agir. Mostraremos o estado:**

**Console.WriteLine("=================");**

**foreach (var e in contexto.ChangeTracker.Entries())**

**{**

**Console.WriteLine(e.State);**

**}**

**Como não modificamos as entidades, ao rodar a aplicação veremos como resultado que os dados estão como *Unchanged*. Faremos uma alteração no último item da lista e verificaremos o estado novamente.**

**static void Main(string[] args)**

**{**

**using(var contexto = LojaContext())**

**{**

**var produtos = contexto.Produtos.ToList();**

**foreach (var p in produtos)**

**{**

**Console.WriteLine(p);**

**}**

**Console.WriteLine("=================");**

**foreach (var e in contexto.ChangeTracker.Entries())**

**{**

**Console.WriteLine(e);**

**}**

**var p1 = produtos.Last();**

**p1.Nome = "007 - O Espiao Que Me Amava";**

**Console.WriteLine("=================");**

**foreach (var e in contexto.ChangeTracker.Entries())**

**{**

**Console.WriteLine(e);**

**}**

**//contexto.SaveChages();**

**//Console.WriteLine("=================");**

**//produtos = contexto.Produtos.ToList();**

**//foreach (var p in produtos)**

**//{**

**// Console.WriteLine(p);**

**//}**

**}**

**}**

**Rodando a aplicação novamente vemos que após a mudança, o último elemento está como *Modified*. É dessa forma que o Entity gerencia as entidades. Quando usamos o SaveChanges, o Entity emite um comando SQL diferente para cada estado.**

**Nossas classes não tem nenhuma lógica adicional para facilitar o monitoramento de mudanças em suas propriedades. Isto é, elas não herdam de nenhuma classe com esse propósito, e não possuem nenhuma propriedade que registra se foram modificadas (por exemplo,*IsDirty*). São classes simples que usam o padrão POCO (Plain Old C# Objects).**

**Então, como o ChangeTracker sabe que, quando uma propriedade foi alterada, ele deve fazer um UPDATE no banco?**

**O Entity guarda um *snapshot* dos valores dos objetos por padrão. Quando aquele objeto começa a ser monitorado pelo Entity, seja através de métodos que recuperam objetos do banco via SELECT (por exemplo ToList, First, Find, etc.), seja através do método Entry que cria uma entrada no ChangeTracker para o objeto passado como argumento do método.**

**E chama o método DetectChanges ao executar o SaveChanges. O que esse método faz? DetectChanges verifica diferenças entre os valores atuais das propriedades da entidade e os valores originais guardados no *snapshot* quando ela foi anexada ao contexto.**

**É possível desligar o monitoramento automático de mudanças através de uma propriedade *booleana* no ChangeTracker chamada AutoDetectChangesEnabled. Quando isso é necessário? Quando você tiver uma gravação massiva de objetos através do SaveChanges, a performance pode sofrer impacto considerável, uma vez que o método DetectChanges será chamado e o ChangeTracker irá percorrer toda a lista de objetos sendo monitorados.**

**vimos que quando colocamos o Entity para cuidar das entidades, ele passa a monitorar o estado desses objetos. Quando buscamos dados no banco, o estado veio como *Unchanged*, ao alterarmos alguma informação do dado, ele passou a ser *Modified*.**

**O que faremos durante essa parte, não é necessário que você repita. O intuito é mostrar o SQL que o Entity está gerando e qual a relação desse SQL com o estado atual dos objetos.**

**Primeiro vamos configurar alguns detalhes na class Program. Dentro do método Main() vamos adicionar o provedor de serviços GetInfrasctructure<IServiceProvider>(), lembrando de adicionar o *namespaces*.**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**using (var contexto = new LojaContext())**

**{**

**var serviceProvider = contexto.GetInfrasctructure<IServiceProvider>();**

**// ...**

**}**

**}**

**}**

**Pediremos para o serviceProvider fornecer um serviço específico que cria *Loggers*.**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**using (var contexto = new LojaContext())**

**{**

**var serviceProvider = contexto.GetInfrastructure<IServiceProvider>();**

**var loggerFactory = serviceProvider.GetService<ILoggerFactory>();**

**// ...**

**}**

**}**

**}**

**Com o loggerFactory passaremos para o método AddProvider() um *Logger*específico que chamaremos de SqlLoggerProvider. O SqlLoggerProvider chama um método chamado Create().**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**using (var contexto = new LojaContext())**

**{**

**var serviceProvider = contexto.GetInfrasctructure<IServiceProvider>();**

**var loggerFactory = serviceProvider.GetService<ILoggerFactory>();**

**loggerFactory.AddProvider(SqlLoggerProvider.Create());**

**// ...**

**}**

**}**

**}**

**Porém a classe SqlLoggerProvider não existe, vamos criá-la.**

**Lembrando que todos esses passos não são necessários que sejam feitos, é apenas para mostrar como o SQL que o Entity está gerando.**

**Criaremos a classe código SqlLoggerProvider e deixaremos da seguinte maneira:**

**using Microsoft.EntityFrameworkCore.Storage;**

**using Microsoft.Extensions.Logging;**

**using System;**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**public class SqlLoggerProvider : ILoggerProvider**

**{**

**public static ILoggerProvider Create()**

**{**

**return new SqlLoggerProvider();**

**}**

**public ILogger CreateLogger(string categoryName)**

**{**

**if (categoryName == typeof(IRelationalCommandBuilderFactory).FullName)**

**{**

**return new SqlLogger();**

**}**

**return new NullLogger();**

**}**

**public void Dispose()**

**{**

**}**

**}**

**internal class NullLogger : ILogger**

**{**

**public IDisposable BeginScope<TState>(TState state)**

**{**

**return null;**

**}**

**public bool IsEnabled(LogLevel logLevel)**

**{**

**return true;**

**}**

**public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception exception, Func<TState, Exception, string> formatter)**

**{**

**//não faz nada**

**}**

**}**

**public class SqlLogger : ILogger**

**{**

**public IDisposable BeginScope<TState>(TState state)**

**{**

**return null;**

**}**

**public bool IsEnabled(LogLevel logLevel)**

**{**

**return true;**

**}**

**public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception exception, Func<TState, Exception, string> formatter)**

**{**

**Console.WriteLine("");**

**Console.WriteLine(formatter(state, exception));**

**Console.WriteLine("");**

**}**

**}**

**}**

**Executando a aplicação veremos o SQl:**

**SELECT [p].[Id], [p].[Categoria], [p].[Nome], [p].[Preco]**

**FROM [Produtos] AS [p]**

**Mostrando também o estado de cada dado. Como o último item da lista estamos modificando o nome e não salvamos, o seu estado é *Modified*. O que acontecerá quando salvarmos essas alterações?**

**Tiraremos os comentários do comando contexto.SaveChanges(), que está depois da alteração que fizemos no nome do último elemento da lista.**

**static void Main(string[] args)**

**{**

**using(var contexto = LojaContext())**

**{**

**var serviceProvider = contexto.GetInfrasctructure<IServiceProvider>();**

**var loggerFactory = serviceProvider.GetService<ILoggerFactory>();**

**loggerFactory.AddProvider(SqlLoggerProvider.Create());**

**var produtos = contexto.Produtos.ToList();**

**foreach (var p in produtos)**

**{**

**Console.WriteLine(p);**

**}**

**Console.WriteLine("=================");**

**foreach (var e in contexto.ChangeTracker.Entries())**

**{**

**Console.WriteLine(e);**

**}**

**var p1 = produtos.Last();**

**p1.Nome = "007 - O Espiao Que Me Amava";**

**Console.WriteLine("=================");**

**foreach (var e in contexto.ChangeTracker.Entries())**

**{**

**Console.WriteLine(e);**

**}**

**contexto.SaveChages();**

**//Console.WriteLine("=================");**

**//produtos = contexto.Produtos.ToList();**

**//foreach (var p in produtos)**

**//{**

**// Console.WriteLine(p);**

**//}**

**}**

**}**

**Executando a aplicação novamente vemos que após apresentar o a mensagem dos estados, foi gerada um SQL para atualizar no banco de dados.**

**UPDATE [Produtos] SET [Nome] = @p0**

**WHERE [Id] = @p1;**

**Então, quando chamamos o contexto.SaveChages(), ele olha para cada objeto e verifica o seu estado. Caso seja um estado que é preciso sincronizar com o banco, ele executará o SQL. Para o estado "Modified", ele executa o SQL Update.**

**Aprendemos o que acontece com um dado quando é alterado, o que veremos agora é como o *ChangeTracker* age ao adicionar um produto.**

**Comentaremos o trecho de código onde alteramos o último produto. Em seguida adicionaremos o um novo produto, e para colocarmos sobre o controle do Entity chamaremos o método contexto.Produtos.Add();**

**static void Main(string[] args)**

**{**

**using(var contexto = new LojaContext())**

**{**

**var serviceProvider = contexto.GetInfrasctructure<IServiceProvider>();**

**var loggerFactory = serviceProvider.GetService<ILoggerFactory>();**

**loggerFactory.AddProvider(SqlLoggerProvider.Create());**

**var produtos = contexto.Produtos.ToList();**

**foreach (var p in produtos)**

**{**

**Console.WriteLine(p);**

**}**

**Console.WriteLine("=================");**

**foreach (var e in contexto.ChangeTracker.Entries())**

**{**

**Console.WriteLine(e);**

**}**

**//var p1 = produtos.Last();**

**//p1.Nome = "007 - O Espiao Que Me Amava";**

**var novoProduto = new Produto()**

**{**

**Nome = "Desinfetante",**

**Categoria = "Limpeza",**

**Preco = 2.99;**

**};**

**contexto.Produtos.Add(novoProduto);**

**Console.WriteLine("=================");**

**foreach (var e in contexto.ChangeTracker.Entries())**

**{**

**Console.WriteLine(e);**

**}**

**contexto.SaveChages();**

**//Console.WriteLine("=================");**

**//produtos = contexto.Produtos.ToList();**

**//foreach (var p in produtos)**

**//{**

**// Console.WriteLine(p);**

**//}**

**}**

**}**

**Para evitarmos a repetição de código, vamos extrair o laço onde mostramos o estado das entidade no *ChangeTracker*.**

**private static void ExibeEntries(IEnumerable<EntityEntry> entries)**

**{**

**Console.WriteLine("===================");**

**foreach (var e in entries)**

**{**

**Console.WriteLine(e);**

**}**

**}**

**Além do estado, mostraremos também as informações do produto.**

**private static void ExibeEntries(IEnumerable<EntityEntry> entries)**

**{**

**Console.WriteLine("===================");**

**foreach (var e in entries)**

**{**

**Console.WriteLine(e.Entity.ToString() + " - " + e.State);**

**}**

**}**

**Na classe Produto dentro do ToString(), colocaremos todas as informações do produto.**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**internal class Produto**

**{**

**// ...**

**public override string ToString()**

**{**

**return $"Produto: {this.Id}, {this.Nome}, {this.Categoria}, {this.Preco}";**

**}**

**}**

**}**

**Vamos substituir os laços de repetição por uma chamada do método ExibeEntries(). A classe Program ficará da seguinte maneira:**

**namespace Alura.Loja.Testes.ConsoleApp**

**{**

**static void Main(string[] args)**

**{**

**using(var contexto = new LojaContext())**

**{**

**var serviceProvider = contexto.GetInfrasctructure<IServiceProvider>();**

**var loggerFactory = serviceProvider.GetService<ILoggerFactory>();**

**loggerFactory.AddProvider(SqlLoggerProvider.Create());**

**var produtos = contexto.Produtos.ToList();**

**ExibeEntries(contexto.ChangeTracker.Entries());**

**var novoProduto = new Produto**

**{**

**Nome = "Desinfetante",**

**Categoria = "Limpeza",**

**Preco = 2.99;**

**};**

**contexto.Produtos.Add(novoProduto);**

**ExibeEntries(contexto.ChangeTracker.Entries());**

**contexto.SaveChages();**

**ExibeEntries(contexto.ChangeTracker.Entries());**

**}**

**}**

**private static void ExibeEntries(IEnumerable<EntityEntry> entries)**

**{**

**Console.WriteLine("===================");**

**foreach (var e in contexto.ChangeTracker.Entries())**

**{**

**Console.WriteLine(e.Entity.ToString() + " - " + e.State);**

**}**

**}**

**}**

**Após executar a aplicação, veremos os resultados. O primeiro resultado é o comando SQL select. Em seguida foi mostrado todos os dados cadastrados no banco, com o estado valor *Unchanged*.**

**O produto "Desinfetante" que adicionamos, está com o menor valor inteiro no Ide com o estado *Added*. Após ser salvo com o SaveChanges(), o resultado é o comando SQL insert:**

**INSERT INTO [Produtos] ([Categoria], [Nome], [Preco])**

**VALUES (@p0, @p1, @p2);**

**Novamente um select será gerado trazendo todos os dados cadastrados no banco. Agora o produto "Desinfetante" está com o Id gerado pelo banco e o estado *Unchanged*.**

**Você já deve estar imaginando que é por meio do método do Entity contexto.Produtos.Remove(). Faremos alguns exemplos para ver como o ChangeTracker irá se comportar ao chamar o método contexto.Produtos.Remove().**

**Comentaremos o código onde adicionamos um produto ao contexto e onde salvamos as alterações. Em seguida, pegaremos um produto do banco de dados e armazenaremos em uma variável chamada p1, e passamos como argumento para o contexto.Produtos.Remove().**

**static void Main(String [] args)**

**{**

**using(var contexto = LojaContext())**

**{**

**var serviceProvider = contexto.GetInfrasctructure<IServiceProvider>();**

**var loggerFactory = serviceProvider.GetService<ILoggerFactory>();**

**loggerFactory.AddProvider(SqlLoggerProvider.Create());**

**var produtos = contexto.Produtos.ToList();**

**ExibeEntries(contexto.ChangeTracker.Entries());**

**//var novoProduto = new Produto**

**//{**

**// Nome = "Desinfetante",**

**// Categoria = "Limpeza",**

**// Preco = 2.99;**

**//};**

**//contexto.Produtos.Add(novoProduto);**

**var p1 = produtos.First();**

**contexto.Produtos.Remove(p1);**

**ExibeEntries(contexto.ChangeTracker.Entries());**

**//contexto.SaveChages();**

**//ExibeEntries(contexto.ChangeTracker.Entries());**

**}**

**}**

**Por enquanto não usaremos o contexto.SaveChanges() para salvar as alterações. Rodaremos a aplicação para ver qual estado o Entity colocou para o primeiro produto da lista.**

**No resultado, vemos que foi executado o comando SQL SELECT para trazer as informações do banco. Após isso, foram apresentados todos os dados com o estado *Unchanged*. No código removemos o primeiro produto usando o contexto.Produtos.Remove(), dessa forma, o produto foi apresentado com o estado *Deleted*.**

**Tiraremos os comentários do contexto.SaveChanges() e do ExibeEntries() que estão após a remoção.**

**static void Main(String [] args)**

**{**

**using(var contexto = LojaContext())**

**{**

**var serviceProvider = contexto.GetInfrasctructure<IServiceProvider>();**

**var loggerFactory = serviceProvider.GetService<ILoggerFactory>();**

**loggerFactory.AddProvider(SqlLoggerProvider.Create());**

**var produtos = contexto.Produtos.ToList();**

**ExibeEntries(contexto.ChangeTracker.Entries());**

**//var novoProduto = new Produto**

**//{**

**// Nome = "Desinfetante",**

**// Categoria = "Limpeza",**

**// Preco = 2.99;**

**//};**

**//contexto.Produtos.Add(novoProduto);**

**var p1 = produtos.First();**

**contexto.Produtos.Remove(p1);**

**ExibeEntries(contexto.ChangeTracker.Entries());**

**contexto.SaveChages();**

**ExibeEntries(contexto.ChangeTracker.Entries());**

**}**

**}**

**Executando a aplicação, notamos que após apresentar os dados e colocar o estado do primeiro dado como *Deleted*, o Entity executou um comando SQL DELETE.**

**DELETE FROM [Produtos]**

**WHERE [Id] = @p0;**

**Ao mostrar novamente os dados, o primeiro produto da lista já não está sendo controlado pelo Entity e foi removido do banco de dados. Mas o que aconteceria se adicionarmos um produto no Entity e removê-lo antes do contexto.SaveChanges()? Ele enviaria um comando SQL mesmo sem ter o produto no banco de dados?**

**static void Main(String [] args)**

**{**

**using(var contexto = LojaContext())**

**{**

**var serviceProvider = contexto.GetInfrasctructure<IServiceProvider>();**

**var loggerFactory = serviceProvider.GetService<ILoggerFactory>();**

**loggerFactory.AddProvider(SqlLoggerProvider.Create());**

**var produtos = contexto.Produtos.ToList();**

**ExibeEntries(contexto.ChangeTracker.Entries());**

**var novoProduto = new Produto()**

**{**

**Nome = "Sabão em pó",**

**Categoria = "Limpeza",**

**Preco = 5.99**

**};**

**contexto.Produtos.Add(novoProduto);**

**contexto.Produtos.Remove(novoProduto);**

**ExibeEntries(contexto.ChangeTracker.Entries());**

**contexto.SaveChanges();**

**ExibeEntries(contexto.ChangeTracker.Entries());**

**}**

**}**

**Nesse caso, foi executado o comando SQL SELECT para trazer os produtos com o estado *Unchanged*, porém após adicionar e remover, o produto "Sabão em pó" não apareceu como um produto *Added*. Isso aconteceu porque após removermos, o Entity retira o produto da sua lista de rastreamento. Com isso, após salvarmos as alterações, não é enviada nenhum comando SQL para o banco.**

**Mas o que aconteceu com esse produto novoProduto? Podemos ver o seu estado usando o método contexto.Entry() que nos retorna uma variável do tipo EntityEntry. Adicionaremos ao final do método Main():**

**var entry = contexto.Entry(novoProduto);**

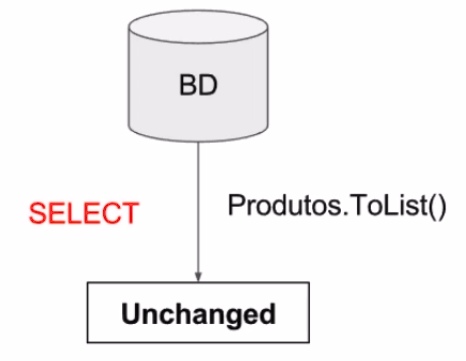
**Console.WriteLine("\n\n" + entry.Entity.ToString() + " - " + entry.State);**

**Rodando a aplicação podemos ver que o estado desse produto está como *Detached*. Este estado representa que o objeto não está sendo monitorado.**

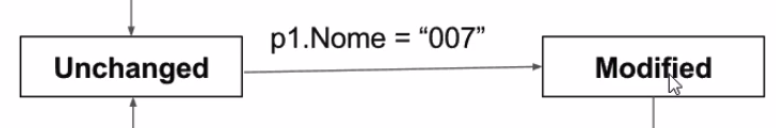
**Detached**

**Correto! Como o objeto já estava sendo monitorado pois o método Add() foi utilizado , ao utilizarmos o método Remove() ele vai para o estado de Detacheddiretamente, pois ele ainda não foi para o banco.**

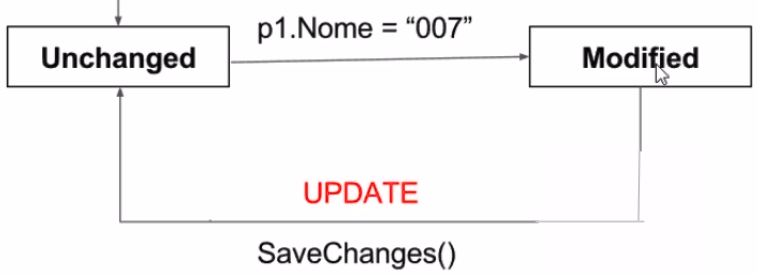
**Recapitularemos o que fizemos durante a aula. Quando usamos o método Produtos.ToList(), fizemos um comando SQL SELECT no banco e recebemos os produtos com o estado *Unchanged***

******

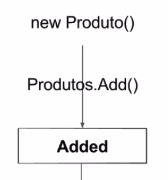
**Após pegarmos o objeto *Unchanged* e alterarmos algum atributo, ele passa para o estado *Modified*.**

****

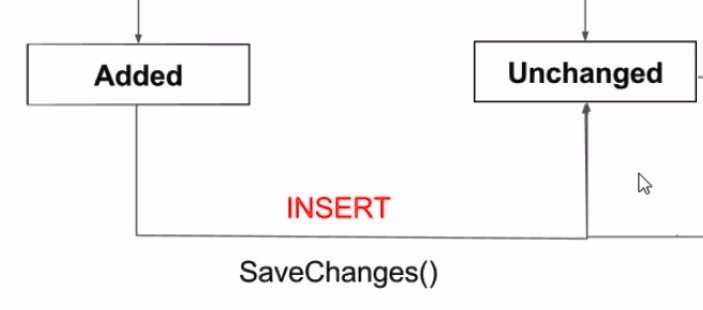
**Depois de usarmos o método SaveChanges(), é executado um comando SQL UPDATE, modificado o estado do produto para *Unchanged*.**

****

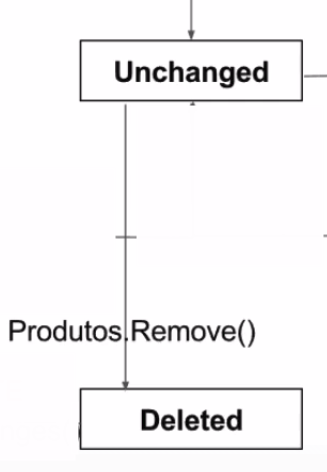
**Quando criamos um produto e adicionamos ao contexto do Entity com o método Produtos.Add(), o estado do produto fica *Added*.**

****

**Depois de usarmos novamente o método SaveChanges(), é executado um comando SQL INSERT alterando o estado do produto para *Unchaged*.**

****

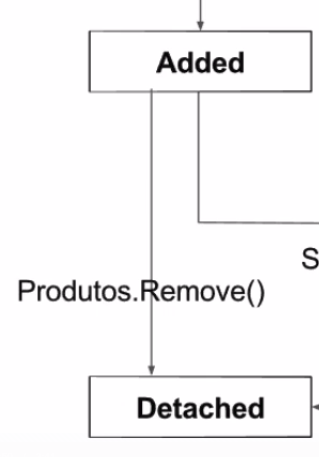
**Quando usamos o Produtos.Remove() em um objeto, ele passa para o estado *Deleted*.**

****

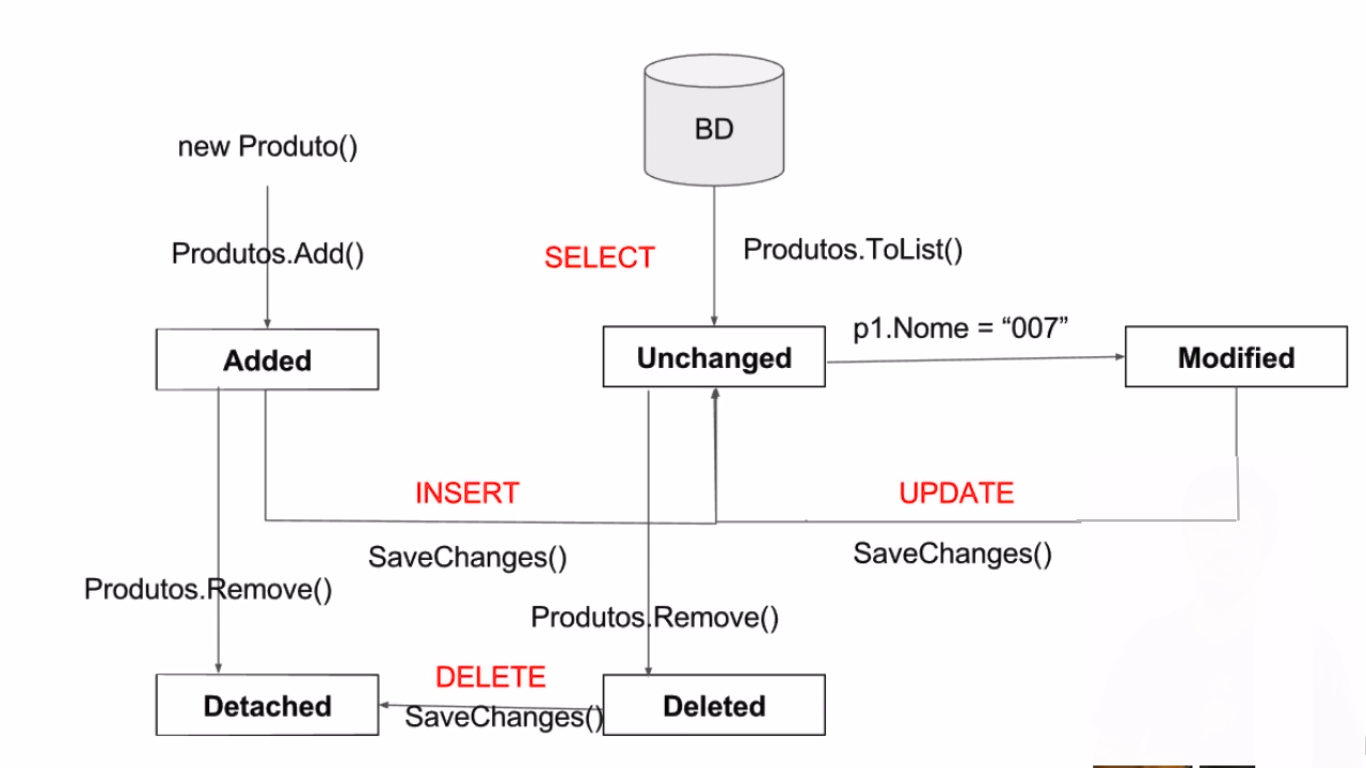
**Quando salvamos as alterações ele passa a ser *Detached*, que representa um objeto que não é mais rastreado pelo Entity. E claro, o produto também é removido do banco de dados.**

****

**Quando adicionamos um produto e ele é removido sem ter sido salvo com o SaveChanges(), ele passa de *Added* para *Detached*.**

****

**O diagrama fica da seguinte forma:**

****

**Segue abaixo um resumo dos estados e seu comportamento.**

###Added O objeto é novo, foi adicionado ao contexto, e o método SaveChangesainda não foi executado. Depois que as mudanças são salvas, o estado do objeto muda para **Unchanged**. Objetos no estado **Added** não têm seus valores rastreados em sua instância de EntityEntry.

###Deleted O objeto foi excluído do contexto. Depois que as mudanças foram salvas, seu estado muda para **Detached**.

###Detached O objeto existe, mas não está sendo monitorado. Uma entidade fica nesse estado imediatamente após ter sido criada e antes de ser adicionada ao contexto. Ela também fica nesse estado depois que foi removida do contexto através do método Detach ou se é carregada por um método com opção NoTracking. Não existem instâncias de EntityEntry associadas a objetos com esse estado.

###Modified Uma das propriedades escalares do objeto foi modificada e o método SaveChanges ainda não foi executado. Quando o monitoramento automático de mudanças está desligado, o estado é alterado para **Modified**apenas quando o método DetectChanges é chamado. Quando as mudanças são salvas, o estado do objeto muda para **Unchanged**.

###Unchanged O objeto não foi modificado desde que foi anexado ao contexto ou desde a última vez que o método SaveChanges foi chamado.

É muito difícil uma aplicação Web manter-se por bastante tempo sem sofrer evoluções. Um exemplo mesmo é o sistema de loja que usamos no curso, estamos registrando apenas os produtos, porém também precisamos registrar as vendas dos produtos. Discutiremos como o Entity ajuda a evoluir a aplicação.

Como já fizemos os testes e entendemos como o ***Change Tracker*** funciona, na classe Programpodemos apagar tudo deixando apenas o método Main().

namespace Alura.Loja.Testes.ConsoleApp

{

class Program

{

static void Main(string[] args)

{

}

}

}

A primeira evolução que iremos fazer na aplicação é na classe Produto. O produto tem a propriedade Preco, porém ele não representa o preço total da compra, e sim o **valor unitário**. Por esse motivo mudaremos a propriedade para PrecoUnitario.

internal class Produto

{

public int Id { get; internal set; }

public string Nome { get; internal set; }

public string Categoria { get; internal set; }

public double PrecoUnitario { get; internal set; }

public override string ToString()

{

return $"Produto: {this.Id}, {this.Nome}, {this.Categoria}, {this.PrecoUnitario}";

}

}

Se tentarmos executar a aplicação, teremos diversos erros de compilação. Se analisarmos os pontos de erro, estarão na classe ProdutoDAO, que usa o ADO.Net. Essa é uma das vantagens do Entity, mesmo após alterarmos a aplicação, ele está preparado para essas situações.

Como não estamos mais usando a classe ProdutoDAO iremos removê-la. No **Gerenciador de Soluções**, clicaremos com o botão direito em cima do nome da classe ProdutoDAO, em seguida selecionaremos a opção "Excluir".

Agora que a aplicação voltou a compilar, vamos criar um nova propriedade na classe Produtochamada Unidade. Essa propriedade vai armazenar a unidade do produto, por exemplo *quilos*, *litros*e assim por diante.

internal class Produto

{

public int Id { get; internal set; }

public string Nome { get; internal set; }

public string Categoria { get; internal set; }

public double PrecoUnitario { get; internal set; }

public string Unidade { get; internal set; }

public override string ToString()

{

return $"Produto: {this.Id}, {this.Nome}, {this.Categoria}, {this.PrecoUnitario}";

}

}

Porém como essa mudanças vão refletir no banco de dados? Olhando a nossa tabela, vemos que ela possui apenas Id, Nome, Categoria e Preco. Nós criamos a tabela com base nas informações que estavam contidas no arquivo ddl-produtos.txt.

A sigla ***DDL*** significa ***Data Definition Language***, ela é um subconjunto da linguagem SQL. Durante as aulas falamos bastante dos comandos INSERT, DELETE, UPDATE e SELECT, esses comandos são ***DML*** que significa ***Data Manipulation Language***.

Para ajustarmos a tabela, teríamos que fazer um ALTER TABLE para mudarmos o nome de uma coluna e adicionar uma nova. Porém não queremos mais ter que cuidar da tabela, queremos que o Entity faça essa função, de sincronizar o banco com a classe modelo.

Essa parte responsável pela de sincronização é chamada de ***Migrations***, que fica em outro pacote. Para instalá-lo, vamos em "Ferramentas > Gerenciador de Pacote do NuGet > Console do Gerenciador de Pacotes", e digitamos:

PM> Install-Package Microsoft.EntityFrameworkCore.Tools -Version 1.1.1

Usamos a versão 1.1.1, porém você pode utilizar uma versão mais atualizada!

Microsoft.EntityFrameworkCore.Tools

Temos o banco de dados com a tabela **Produtos**, que contém as colunas **Id** como chave primária, **Nome**, **Categoria** e **Preco**. A classe Produto tem as propriedades Id, Nome, Categoria, PrecoUnitario e Unidade. A classe não está sincronizada com a tabela, por isso usaremos o pacote de ***Migrations*** (Migrações) que instalamos.

As migrações serão feitas no Entity e para isso usaremos comandos. Com o **Console de Gerenciador de Pacotes** aberto, digitaremos o comando para abrir o manual do EntityFramework, assim poderemos ver todos os comandos disponíveis. No console digite:

Get-Help EntityFramework

Ao executar o comando, é apresentado no Console o símbolo, e descrição e a lista de comandos disponíveis.

* Add-Migration
* Remove-Migration
* Update-Database
* Script-Migration
* Drop-Database
* Scaffold-DbContext

A migração é feita em dois passos. O primeiro passo é executarmos o comando Add-Migration.

Já o segundo passo pode ser feitas de duas maneiras diferentes, sendo a primeira gerando um *script*de linguagem DDL com o comando Script-Migration. Esse cenário é mais utilizado quando existe uma equipe de banco de dados separada da equipe de desenvolvimento.

A outra maneira é usarmos o comando Update-Database, onde o Entity pega a nova versão que foi registrada e executa diretamente no banco de dados. Vamos utilizar essa segunda forma.

Começaremos executando o comando para adicionar a migração, passando o nome para ela. Como estamos adicionando uma nova coluna Unidade, daremos o nome da migração de **Unidade**. O comando ficará:

Add-Migration Unidade

Não obtivemos nenhum resultado relevante no Console, mas olhando o projeto podemos ver que uma pasta chamada ***Migrations*** foi criada, contendo duas classes.

A classe que usaremos possui o nome do **arquivo** com um ***Timestamp***, que é a data e hora que o arquivo foi gerado. O nome do arquivo ainda tem a separação por *underscore* e escrito **Unidade**.

Abrindo o arquivo que possui o *Timestamp*, vemos que é a classe Unidade. A classe Unidade herda de Migration, que nos fornece uma API que fará essa sincronização. Também possui dois métodos, Up que serve para atualizar para a versão mais nova das tabelas, e o Down que serve para voltar para uma versão anterior.

Executaremos o comando para atualizar a tabela, passando o parâmetro -Verbose para que o Console apresente todas as operações. O comando ficará da seguinte maneira:

Update-Database -Verbose

Após executar o comando, receberemos um erro informando que a tabela Produtos já existe no banco de dados:

*There is already an object named 'Produtos' in the database*

Porém o Entity criou uma tabela chamada \_\_EFMigrationHistory, que é utilizada para manter o histórico de migrações e o Entity utilizará para controlar as versões executadas no banco de dados.

Como poderemos aplicar essa migração? Precisamos refletir essa alteração no banco, mas para isso precisaríamos da versão onde a tabela foi criada. Veremos como resolver esse problema no próximo vídeo.

Para descobrir quais comandos estão disponíveis no recurso de *Migrations*, use este:

Get-Help EntityFramework

orém quando tentamos executar a migração, recebemos um erro informando que a tabela já existia.

Esse problema foi porque colocamos a criação da tabela e a evolução na mesma execução. Para resolvermos, nós teríamos que separar a criação da evolução da tabela.

Começaremos excluindo a tabela \_\_EFMigrationsHistory do banco, e também a pasta Migrations do projeto. Além disso, voltaremos a classe produto para a sua forma inicial:

namespace Alura.Loja.Testes.ConsoleApp

{

internal class Produto

{

public int Id { get; internal set; }

public string Nome { get; internal set; }

public string Categoria { get; internal set; }

public double Preco { get; internal set; }

public override string ToString()

{

return $"Produto: {this.Id}, {this.Nome}, {this.Categoria}, {this.Preco}";

}

}

}

Executaremos o comando:

PM> Add-Migration Inicial

Agora que já criamos a *Migration* inicial, voltaremos a colocar a evolução da classe, que ficará da seguinte maneira:

namespace Alura.Loja.Testes.ConsoleApp

{

internal class Produto

{

public int Id { get; internal set; }

public string Nome { get; internal set; }

public string Categoria { get; internal set; }

public double PrecoUnitario { get; internal set; }

public string Unidade { get; set; };

public override string ToString()

{

return $"Produto {this.Id}, {this.Nome}, {this.Categoria}, {this.PrecoUnitario}";

}

}

}

Criaremos um nova migração que chamaremos de **Unidade**. O comando será:

Add-Migration Unidade

Acessando a classe de migração Unidade, veremos que ela está bem diferente, refletindo apenas as informações de evolução da tabela. Dentro do método Up() temos a instrução de renomear a coluna Preco para PrecoUnitario, e adicionar uma nova coluna Unidade. Já no método Down, temos a instrução de deletar a coluna Unidade e renomear PrecoUnitario para Preco.

Tentando executar o comando Update-Database receberemos o mesmo erro informando que a tabela já foi criada. A função da tabela \_\_EFMigrationsHistory é apenas registrar quais migrações foram executadas no banco de dados. Ao acessá-la, veremos que nenhuma migração foi executada.

Levando isso em consideração, ao tentarmos efetuar a migração, ele executará todas as migrações que estão na pasta Migrations, inclusive a migração **Inicial**.

Fingiremos para o banco que a migração **Inicial** já foi executada. Entraremos da classe de migração Inicial e comentaremos todo o conteúdo do método Up(), em seguida executaremos Update-Database Inicial. Isso executará apenas a migração com o nome *Inicial* e irá criar uma linha na tabela de histórico de migração.

Agora podemos executar tirar os comentários da classe Inicial e executar o comando Update-Database, dessa vez sem passarmos o nome da classe. A sincronização foi efetuada com sucesso.

Acessando o banco de dados, veremos que a coluna Preco foi renomeada para PrecoUnitario e a coluna Unidade foi criada.

Depois que sua aplicação estiver madura o bastante para ser promovida, surge a questão: como atualizar o banco de dados daquele ambiente específico? Vimos que em organizações com políticas de acesso mais restritas a ambientes críticos, a solução é gerar um arquivo com o script das migrações e entregar esse arquivo à equipe responsável. Essa tarefa é realizada com o comando Script-Migration.

Além disso, também é possível fazer que sua própria aplicação cuide da migração das versões. Ou seja, podemos escrever código em nossa aplicação para que o banco de dados seja sincronizado. Isso é feito através do método de extensão Migrate, que está acessível na propriedade Database da classe DbContext. Essa propriedade representa a instância do banco de dados apontado pelo contexto Entity específico de sua aplicação (no nosso exemplo, LojaContext), e expõe métodos que permitem gerenciar o banco apontado pelo contexto, como por exemplo sua criação, exclusão e validação de existência.

O método Migrate só pode ser usado em bancos de dados relacionais e fica disponível no pacote Microsoft.EntityFrameworkCore.Relational.

Assim, para garantir que todas as migrações estarão aplicadas no banco de dados, podemos escrever:

using(var contexto = new LojaContext())

{

contexto.Database.Migrate();

}

Você precisa garantir que esse código seja executado antes de qualquer acesso aos objetos gerenciados pelo contexto. Isso vai depender do tipo de aplicação que será implementada.

o Entity sincroniza estrutura das classes entidade, com a estrutura das tabelas no banco de dados. Isso era necessário porque precisávamos evoluir a aplicação.

Para uma loja prosperar, é necessário que os clientes comprem. Nessa aula, modelaremos o conceito de compra, em uma nova classe e relacionaremos com o produto. Dessa forma veremos como o Entity trabalhará com relacionamentos.

Na classe Program dentro do método Main(), simularemos uma compra de um produto. Começaremos criando um novo produto e popularemos:

class Program

{

static void Main(string[] args)

{

//compra de 6 pães franceses

var paoFrances = new Produto();

paoFrances.Nome = "Pão Francês";

paoFrances.PrecoUnitario = 0.40;

paoFrances.Unidade = "Unidade";

paoFrances.Categoria = "Padaria";

}

}

Mas como simularemos uma compra? Instanciaremos a classe Compra, populando os atributos Quantidade, Produto e Preco. O valor de Preco é a multiplicação entre o PrecoUnitario do produto e a Quantidade da compra.

class Program

{

static void Main(string[] args)

{

//compra de 6 pães franceses

var paoFrances = new Produto();

paoFrances.Nome = "Pão Francês";

paoFrances.PrecoUnitario = 0.40;

paoFrances.Unidade = "Unidade";

paoFrances.Categoria = "Padaria";

var compra = new Compra();

compra.Quantidade = 6;

compra.Produto = paoFrances;

compra.Preco = paoFrances.PrecoUnitario \* compra.Quantidade;

}

}

Como a classe e as propriedades ainda não foram criadas, podemos usar o atalho "Ctrl + . (ponto)" do Visual Studio, e pedir a criação. Podemos persistir as informações da compra no banco de dados, porém ainda não possuímos a tabela que representa a classe Compra. Para sincronizar a classe com o banco, usaremos novamente ***Migrations***.

ntes de criarmos a migração para representar a classe, entraremos na classe Compra e adicionaremos uma propriedade Id. Essa propriedade irá representar a chave primária da compra. Também removeremos o construtor da classe, que foi gerado automaticamente pelo Visual Studio.

namespace Alura.Loja.Testes.ConsoleApp

{

internal class Compra

{

public int Id { get; set; }

public int Quantidade { get; internal set; }

public Produto Produto { get; internal set; }

public double Preco { get; internal set; }

}

}

Abriremos o **Console do Gerenciador de Pacotes** do NuGet, e digitaremos o comando para adicionarmos a migração:

PM> Add-Migration Compra

Abriremos a classe de migração Compra. Podemos notar que os métodos Up() e Down() estão vazios. O que queremos é criar a tabela **Compra** no banco de dados, mas o que fizemos de errado?

Estamos esquecendo de um passo importante, que é dizer pro Entity gerenciar a persistência da classe Compra. Para fazermos isso, acessaremos a classe de contexto LojaContext, e criaremos uma propriedade DbSet<Compra> Compras.

using Microsoft.EntityFrameworkCore;

using System;

namespace Alura.Loja.Testes.ConsoleApp

{

internal class LojaContext : DbContext

{

public DbSet<Produto> Produtos { get; set; }

public DbSet<Compras> Compras { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=LojaDB;Trusted\_Connection=true;")

}

}

}

Removeremos a migração que acabamos de fazer executando o comando Remove-Migration no Console. A classe de migração com os métodos Up() e Down() vazias, será excluída.

Rodaremos novamente o comando Add-Migration Compra. A classe criada para essa migração possui código nos métodos Up() e Down(). Basicamente o método Up() irá criar uma tabela com as colunas Id, Preco, ProdutoId e Quantidade.

Não criamos nenhuma propriedade na classe Compras com o nome ProdutoId, então de onde veio essa coluna? Isso acontece porque no modelo relacional não é possível guardar uma referência para um objeto, como fazemos no modelo orientado a objetos. Justamente por isso, no modelo relacional é guardado uma *referência para a chave primária* do produto, e o Entity usa na coluna a convenção [nome da classe]Id.

É sempre importante analisarmos o código antes de atualizar as migrações. A linha de criação da coluna ProdutoId permite que um produto seja nulo:

ProdutoId = table.Column<int>(nullable: true);

Quando efetuamos uma compra, é obrigatório ter um produto. Essa migração não está atendendo as nossas necessidades. Por isso iremos removê-la com o comando Remove-Migration.

Para indicar que um Produto é obrigatório, iremos deixar explícito a chave estrangeira ProdutoId.

namespace Alura.Loja.Testes.ConsoleApp

{

internal class Compra

{

public int Id { get; set; }

public int Quantidade { get; internal set; }

public int ProdutoId { get; set; }

public Produto Produto { get; internal set; }

public double Preco { get; internal set; }

}

}

Como um valor do tipo int não pode ser nulo, o Entity irá interpretar que a propriedade deve ser obrigatória. Executaremos o comando Add-Migration Compra para adicionar a migração.

Podemos ver na classe de migração Compra, que a coluna ProdutoId não permite valores nulos:

ProdutoId = table.Column<int>(nullable: false);

Finalmente podemos sincronizar essa migração com o banco de dados, usaremos o comando Update-Database. A tabela Compras foi criada, e a migração foi adicionada no histórico de migrações na tabela \_\_EFMigrationsHistory.

ntes de criarmos a migração para representar a classe, entraremos na classe Compra e adicionaremos uma propriedade Id. Essa propriedade irá representar a chave primária da compra. Também removeremos o construtor da classe, que foi gerado automaticamente pelo Visual Studio.

namespace Alura.Loja.Testes.ConsoleApp

{

internal class Compra

{

public int Id { get; set; }

public int Quantidade { get; internal set; }

public Produto Produto { get; internal set; }

public double Preco { get; internal set; }

}

}

Abriremos o Console do Gerenciador de Pacotes do NuGet, e digitaremos o comando para adicionarmos a migração:

PM> Add-Migration Compra

Abriremos a classe de migração Compra. Podemos notar que os métodos Up() e Down() estão vazios. O que queremos é criar a tabela Compra no banco de dados, mas o que fizemos de errado?

Estamos esquecendo de um passo importante, que é dizer pro Entity gerenciar a persistência da classe Compra. Para fazermos isso, acessaremos a classe de contexto LojaContext, e criaremos uma propriedade DbSet<Compra> Compras.

using Microsoft.EntityFrameworkCore;

using System;

namespace Alura.Loja.Testes.ConsoleApp

{

internal class LojaContext : DbContext

{

public DbSet<Produto> Produtos { get; set; }

public DbSet<Compras> Compras { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=LojaDB;Trusted\_Connection=true;")

}

}

}

Removeremos a migração que acabamos de fazer executando o comando Remove-Migration no Console. A classe de migração com os métodos Up() e Down() vazias, será excluída.

Rodaremos novamente o comando Add-Migration Compra. A classe criada para essa migração possui código nos métodos Up() e Down(). Basicamente o método Up() irá criar uma tabela com as colunas Id, Preco, ProdutoId e Quantidade.

Não criamos nenhuma propriedade na classe Compras com o nome ProdutoId, então de onde veio essa coluna? Isso acontece porque no modelo relacional não é possível guardar uma referência para um objeto, como fazemos no modelo orientado a objetos. Justamente por isso, no modelo relacional é guardado uma referência para a chave primária do produto, e o Entity usa na coluna a convenção [nome da classe]Id.

É sempre importante analisarmos o código antes de atualizar as migrações. A linha de criação da coluna ProdutoId permite que um produto seja nulo:

ProdutoId = table.Column<int>(nullable: true);

Quando efetuamos uma compra, é obrigatório ter um produto. Essa migração não está atendendo as nossas necessidades. Por isso iremos removê-la com o comando Remove-Migration.

Para indicar que um Produto é obrigatório, iremos deixar explícito a chave estrangeira ProdutoId.

namespace Alura.Loja.Testes.ConsoleApp

{

internal class Compra

{

public int Id { get; set; }

public int Quantidade { get; internal set; }

public int ProdutoId { get; set; }

public Produto Produto { get; internal set; }

public double Preco { get; internal set; }

}

}

Como um valor do tipo int não pode ser nulo, o Entity irá interpretar que a propriedade deve ser obrigatória. Executaremos o comando Add-Migration Compra para adicionar a migração.

Podemos ver na classe de migração Compra, que a coluna ProdutoId não permite valores nulos:

ProdutoId = table.Column<int>(nullable: false);

Finalmente podemos sincronizar essa migração com o banco de dados, usaremos o comando Update-Database. A tabela Compras foi criada, e a migração foi adicionada no histórico de migrações na tabela \_\_EFMigrationsHistory.

criamos uma nova classe chamada Compra e sincronizamos o modelo de classe com o banco usando o recurso de ***Migrations***. Temos tudo que precisamos para registrar a compra do banco de dados.

Na classe Program, iremos instanciar um objeto do contexto de LojaContext. Com o objeto, iremos adicionar a compra ao contexto do Entity e salvar as alterações. Para visualizarmos as ações, colocaremos o código de *Logger*. A classe ficará da seguinte maneira:

class Program

{

static void Main(string[] args)

{

//compra de 6 pães franceses

var paoFrances = new Produto();

paoFrances.Nome = "Pão Francês";

paoFrances.PrecoUnitario = 0.40;

paoFrances.Unidade = "Unidade";

paoFrances.Categoria = "Padaria";

var compra = new Compra();

compra.Quantidade = 6;

compra.Produto = paoFrances;

compra.Preco = paoFrances.PrecoUnitario \* compra.Quantidade;

using(var contexto = new LojaContext())

{

var serviceProvider = contexto.GetInfrastructure<IServiceProvider>();

var loggerFactory = serviceProvider.GetService<ILoggerFactory>();

loggerFactory.AddProvider(SqlLoggerProvider.Create());

contexto.Compras.Add(compra);

contexto.SaveChanges();

}

}

}

Sabemos que o estado de compra vai ser ***Added***, mas qual será o estado do paoFrances? Nós não pegamos o paoFrances do banco de dados, ele é um produto novo. Tivemos todo o trabalho de refazer a **migração** Compra justamente para não ter um produto nulo, como será que o Entity irá reagir nessa situação?

Quando adicionarmos o compra no ***Change Tracker***, o Entity perceberá que existe uma referência ao produto paoFrances e também o incluirá para ser supervisionado.

Comentaremos a linha do SaveChanges(), e colocaremos na classe Program o método ExibeEntries() que usamos nas aulas anteriores.

class Program

{

static void Main (string [] args)

{

//compra de 6 pães franceses

var paoFrances = new Produto();

paoFrances.Nome = "Pão Francês";

paoFrances.PrecoUnitario = 0.40;

paoFrances.Unidade = "Unidade";

paoFrances.Categoria = "Padaria";

var compra = new Compra();

compra.Quantidade = 6;

compra.Produto = paoFrances;

compra.Preco = paoFrances.PrecoUnitario \* compra.Quantidade;

using(var contexto = new LojaContext())

{

var serviceProvider = contexto.GetInfraestructure<IServiceProvider>();

var loggerFactory = serviceProvider.GetService<ILoggerFactory>();

loggerFactory.AddProvider(SqlLoggerProvider.Create());

contexto.Compras.Add(compra);

ExibeEntries(contexto.ChangeTracker.Entries());

//contexto.SaveChanges();

}

}

private static void ExibeEntries(IEnumerable<EntityEntry> entries)

{

foreach(var e in entries)

{

Console.WriteLine(e.Entity.ToString() + " - " + e.State);

}

}

}

Executaremos a aplicação, como resultado veremos que temos duas entidades, um do tipo Compra e outro do tipo Produto. As duas entidades estão com o estado *Added*, por isso o Entity irá gerar o comando SQL INSERT para cada um deles. O contexto foi esperto o suficiente para adicionar a referência do produto contida na compra.

Removeremos o comentário da linha contexto.SaveChanges() e executaremos a aplicação. Veremos que o Entity inseriu primeiro o produto no banco de dados, depois inseriu a compra, justamente por ser necessário ter um produto preenchido para incluirmos a compra. Se olharmos o banco, veremos todas as informações persistidas.

introduzimos a classe Compra, para representarmos a compra de um produto com base no preço e quantidade. Utilizamos essa funcionalidade para introduzir o conceito de relacionamento entre entidades. Nessa aula continuaremos falando sobre relacionamentos, porém em vez de ter relacionamento apenas com a instância de uma única classe, faremos o relacionamento para uma *coleção de instâncias*.

Vamos imaginar que a nossa loja crie regularmente, promoções para baixar o estoque dos produtos. Representaremos esse conceito criando uma classe chamada de Promocao.

namespace Alura.Loja.Testes.ConsoleApp

{

public class Promocao

{

}

}

Agora na classe Program, dentro do método Main() criaremos um cenário para essas promoções. Deixaremos dentro do método apenas a instancia do contexto do Entity e o código do *Logger*.

Instanciaremos uma promoção e armazenaremos em um objeto chamado promocaoDePascoa. A partir desse objeto, chamaremos as propriedades Descricao, DataInicio, DataTermino. Além dessas propriedades, adicionaremos três produtos na propriedade Produtos.

class Program

{

static void Main(string[] args)

{

var promocaoDePascoa = new Promocao();

promocaoDePascoa.Descricao = "Páscoa Feliz";

promocaoDePascoa.DataInicio = DateTime.Now;

promocaoDePascoa.DataTermino = DateTime.Now.AddMonths(3);

promocaoDePascoa.Produtos.Add(new Produto());

promocaoDePascoa.Produtos.Add(new Produto());

promocaoDePascoa.Produtos.Add(new Produto());

using(var contexto = new LojaContext())

{

var serviceProvider = contexto.GetInfra structure<IServiceProvider>();

var loggerFactory = serviceProvider.GetService<ILoggerFactory>();

loggerFactory.AddProvider(SqlLoggerProvider.Create());

}

}

private static void ExibeEntries(IEnumerable<EntityEntry> entries)

{

foreach(var e in entries)

{

Console.WriteLine(e.Entity.ToString() + " - " + e.State);

}

}

}

As propriedades ainda não existem na classe Promocao, por isso iremos criá-la.

namespace Alura.Loja.Testes.ConsoleApp

{

public class Promocao

{

public string Descricao { get; internal set; }

public DateTime DataInicio { get; internal set; }

public DateTime DataTermino { get; internal set; }

public IList<Produto> Produtos { get; internal set; }

}

}

Para podermos usar a propriedade Produtos, é necessário colocarmos a classe Produto como public. Como o Entity irá refletir isso no banco de dados? Sabemos que a classe Promocao ainda não tem a sua tabela representativa, faremos essa configuração.

Na classe LojaContext, criaremos uma nova propriedade DbSet<Promocao> Promocoes.

using Microsoft.EntityFrameworkCore;

using System;

namespace Alura.Loja.Teste.ConsoleApp

{

public class LojaContext : DbContext

{

public DbSet<Produto> Produtos { get; set; }

public DbSet<Compra> Compras { get; set; }

public DbSet<Promocao> Promocoes { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=LojaDB;Trusted\_Connection=true;")

}

}

}

Se tentarmos usar o comando Add-Migration receberemos um erro informando que não possuímos um chave primária, por isso criaremos na classe promoção a propriedade Id. Em seguida, podemos finalmente adicionar a migração que chamaremos de Promocao. O comando completo será Add-Migration Promocao.

namespace Alura.Loja.Testes.ConsoleApp

{

public class Promocao

{

public int Id { get; set; }

public string Descricao { get; internal set; }

public DateTime DataInicio { get; internal set; }

public DateTime DataTermino { get; internal set; }

public IList<Produto> Produtos { get; internal set; }

}

}

Analisando a **classe de migração** Promocao, veremos que ele está criando um coluna PromocaoId na tabela Produto. O Entity interpretou que cada produto está associado a **apenas um promoção**. O que não faz sentido para a nossa aplicação. O que queremos é que a promoção acesse uma coleção de produtos, e o produto acesse uma coleção de promoções. Removeremos essa migração com Remove-Migration.

Para resolvermos isso, adicionaremos na classe Produto uma lista de promoções.

namespace Alura.Loja.Testes.ConsoleApp

{

public class Produto

{

public int Id { get; internal set; }

public int Nome { get; internal set; }

public int Categoria { get; internal set; }

public int PrecoUnitario { get; internal set; }

public string Unidade { get; set; }

public IList<Promocao> Promocoes { get; set; }

public override string ToString()

{

return $"Produto: {this.Id}, {this.Nome}, {this.Categoria}, {this.PrecoUnitario}";

}

}

}

Dessa forma temos o relacionamento de muitas promoções para muitos produtos. Porém se tentarmos adicionar a migração teremos um erro. A versão **6** que é a anterior do Entity Framework Core, conseguia identificar as tabelas JOIN automaticamente. Mas na versão atual, que é um ***subset***da versão anterior, foi deixado de fora (por enquanto) essa função.

para representarmos a compra de um produto com base no preço e quantidade. Utilizamos essa funcionalidade para introduzir o conceito de relacionamento entre entidades. Nessa aula continuaremos falando sobre relacionamentos, porém em vez de ter relacionamento apenas com a instância de uma única classe, faremos o relacionamento para uma *coleção de instâncias*.

Vamos imaginar que a nossa loja crie regularmente, promoções para baixar o estoque dos produtos. Representaremos esse conceito criando uma classe chamada de Promocao.

namespace Alura.Loja.Testes.ConsoleApp

{

public class Promocao

{

}

}

Agora na classe Program, dentro do método Main() criaremos um cenário para essas promoções. Deixaremos dentro do método apenas a instancia do contexto do Entity e o código do *Logger*.

Instanciaremos uma promoção e armazenaremos em um objeto chamado promocaoDePascoa. A partir desse objeto, chamaremos as propriedades Descricao, DataInicio, DataTermino. Além dessas propriedades, adicionaremos três produtos na propriedade Produtos.

class Program

{

static void Main(string[] args)

{

var promocaoDePascoa = new Promocao();

promocaoDePascoa.Descricao = "Páscoa Feliz";

promocaoDePascoa.DataInicio = DateTime.Now;

promocaoDePascoa.DataTermino = DateTime.Now.AddMonths(3);

promocaoDePascoa.Produtos.Add(new Produto());

promocaoDePascoa.Produtos.Add(new Produto());

promocaoDePascoa.Produtos.Add(new Produto());

using(var contexto = new LojaContext())

{

var serviceProvider = contexto.GetInfra structure<IServiceProvider>();

var loggerFactory = serviceProvider.GetService<ILoggerFactory>();

loggerFactory.AddProvider(SqlLoggerProvider.Create());

}

}

private static void ExibeEntries(IEnumerable<EntityEntry> entries)

{

foreach(var e in entries)

{

Console.WriteLine(e.Entity.ToString() + " - " + e.State);

}

}

}

As propriedades ainda não existem na classe Promocao, por isso iremos criá-la.

namespace Alura.Loja.Testes.ConsoleApp

{

public class Promocao

{

public string Descricao { get; internal set; }

public DateTime DataInicio { get; internal set; }

public DateTime DataTermino { get; internal set; }

public IList<Produto> Produtos { get; internal set; }

}

}

Para podermos usar a propriedade Produtos, é necessário colocarmos a classe Produto como public. Como o Entity irá refletir isso no banco de dados? Sabemos que a classe Promocao ainda não tem a sua tabela representativa, faremos essa configuração.

Na classe LojaContext, criaremos uma nova propriedade DbSet<Promocao> Promocoes.

using Microsoft.EntityFrameworkCore;

using System;

namespace Alura.Loja.Teste.ConsoleApp

{

public class LojaContext : DbContext

{

public DbSet<Produto> Produtos { get; set; }

public DbSet<Compra> Compras { get; set; }

public DbSet<Promocao> Promocoes { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=LojaDB;Trusted\_Connection=true;")

}

}

}

Se tentarmos usar o comando Add-Migration receberemos um erro informando que não possuímos um chave primária, por isso criaremos na classe promoção a propriedade Id. Em seguida, podemos finalmente adicionar a migração que chamaremos de Promocao. O comando completo será Add-Migration Promocao.

namespace Alura.Loja.Testes.ConsoleApp

{

public class Promocao

{

public int Id { get; set; }

public string Descricao { get; internal set; }

public DateTime DataInicio { get; internal set; }

public DateTime DataTermino { get; internal set; }

public IList<Produto> Produtos { get; internal set; }

}

}

Analisando a **classe de migração** Promocao, veremos que ele está criando um coluna PromocaoId na tabela Produto. O Entity interpretou que cada produto está associado a **apenas um promoção**. O que não faz sentido para a nossa aplicação. O que queremos é que a promoção acesse uma coleção de produtos, e o produto acesse uma coleção de promoções. Removeremos essa migração com Remove-Migration.

Para resolvermos isso, adicionaremos na classe Produto uma lista de promoções.

namespace Alura.Loja.Testes.ConsoleApp

{

public class Produto

{

public int Id { get; internal set; }

public int Nome { get; internal set; }

public int Categoria { get; internal set; }

public int PrecoUnitario { get; internal set; }

public string Unidade { get; set; }

public IList<Promocao> Promocoes { get; set; }

public override string ToString()

{

return $"Produto: {this.Id}, {this.Nome}, {this.Categoria}, {this.PrecoUnitario}";

}

}

}

Dessa forma temos o relacionamento de muitas promoções para muitos produtos. Porém se tentarmos adicionar a migração teremos um erro. A versão **6** que é a anterior do Entity Framework Core, conseguia identificar as tabelas JOIN automaticamente. Mas na versão atual, que é um ***subset***da versão anterior, foi deixado de fora (por enquanto) essa função.

Para termos a tabela de JOIN, criaremos uma classe que representará esse relacionamento e será reconhecido pelo Entity. A classe se chamará PromocaoProduto.

namespace Alura.Loja.Testes.CosoleApp

{

public class PromocaoProduto

{

}

}

A classe terá as propriedades ProdutoId, Produto, PromocaoId e Promocao:

namespace Alura.Loja.Testes.CosoleApp

{

public class PromocaoProduto

{

public int ProdutoId { get; set; }

public Produto produto { get; set; }

public int PromocaoId { get; set; }

public Promocao Promocao { get; set; }

}

}

Na classe Promocao, mudaremos o tipo da lista IList<Produto> para IList<PromocaoProduto>:

namespace Alura.Loja.Testes.ConsoleApp

{

public class Promocao

{

public int Id { get; set; }

public string Descricao { get; internal set; }

public DateTime DataInicio { get; internal set; }

public DateTime DataTermino { get; internal set; }

public IList<PromocaoProduto> Produtos { get; internal set; }

}

}

Na classe Produto também mudaremos o tipo da lista, de IList<Promocao> para IList<PromocaoProduto>:

namespace Alura.Loja.Testes.ConsoleApp

{

public class Produto

{

public int Id { get; internal set; }

public int Nome { get; internal set; }

public int Categoria { get; internal set; }

public int PrecoUnitario { get; internal set; }

public string Unidade { get; set; }

public IList<PromocaoProduto> Promocoes { get; set; }

public override string ToString()

{

return $"Produto: {this.Id}, {this.Nome}, {this.Categoria}, {this.PrecoUnitario}";

}

}

}

Antes de adicionarmos a migração, é necessário comentarmos o trecho de código onde adicionamos um novo produto na promoção que simulamos na classe Program:

//promocaoDePascoa.Produtos.Add(new Produto());

//promocaoDePascoa.Produtos.Add(new Produto());

//promocaoDePascoa.Produtos.Add(new Produto());

Tentaremos adicionar a migração com o comando Add-Migration Promocao. Receberemos um erro informando que é necessário um **Id** para ser a chave primária da classe PromocaoProduto. Mas será que faz sentido? Não, por ser uma classe de JOIN.

Podemos criar uma **chave primária composta**, que será identificada pelo ProdutoId e PromocaoId. Na classe LojaContext iremos sobrescrever outro método, chamado OnModelCreating().

Esse método é executado no evento de criação do modelo. Podemos configurar informado que a entidade PromocaoProduto **tem a chave composta**, com a composição de ProdutoId e PromocaoId. A configuração ficará da seguinte maneira:

namespace Alura.Loja.Testes.ConsoleApp

{

public class LojaContext : DbContext

{

public DbSet<Produto> Produtos { get; set; }

public DbSet<Compra> Compras { get; set; }

public DbSet<Promocao> Promocoes { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder

.Entity<PromocaoProduto>()

.HasKey(pp => new { pp.PromocaoId, pp.ProdutoId });

base.OnModelCreating(modelBuilder);

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=LojaDB;Trusted\_Connection=true;")

}

}

}

Se executarmos o comando Add-Migration Promocao, a migração será criada com sucesso. Analisando a **classe de migração** Promocao, veremos que o no método Up() será criado as tabelas Promocoes e PromocaoProduto. Detalhe que na tabela JOIN, possui a chave composta, com as chaves estrangeiras de Produtos e Promocoes. Assim se deletarmos um produto ou uma promoção, os registros da tabela PromocaoProduto serão deletados em cascata.

Era isso que queríamos. Executaremos o comando Update-Database para que as evoluções da sistema sejam aplicadas no banco de dados. Também será adicionado na tabela de registros, que essa migração foi executada.

Entity trata o relacionamento **muitos para muitos**, usando o conceito da classe Promocao. Nessa aula aprenderemos o relacionamento de **um para um**.

A ideia é criarmos uma entidade para os clientes que compram na loja. Além disso, também criaremos uma outra entidade para o endereço de entrega.

Antes de começarmos, iremos mover todo o conteúdo do método Main() para um novo método chamado MuitosParaMuitos(). Com o método Main() vazio, começaremos instanciando um objeto da classe Cliente.

class Program

{

static void Main(string[] args)

{

var fulano = new Cliente();

fulano.Nome = "Fulaninho de Tal";

}

private static void MuitosParaMuitos()

{

// ...

}

private static void ExibeEntries(IEnumerable<EntityEntry> entries)

{

// ...

}

}

Como a classe Cliente ainda não existe, iremos criá-la. Lembrando que além das propriedades Nome e EnderecoDeEntrega, criaremos a propriedade Id para persistirmos no banco de dados.

namespace Alura.Loja.Testes.ConsoleApp

{

public class Cliente

{

public int Id { get; set; }

public string Nome { get; internal set; }

public Endereco EnderecoDeEntrega { get; set; }

}

}

Criaremos agora a classe Endereco com as propriedades Numero, Logradouro, Complemento, Bairro e Cidade.

namespace Alura.Loja.Testes.ConsoleApp

{

public class Endereco

{

public int Numero { get; internal set; }

public string Logradouro { get; internal set; }

public string Complemento { get; internal set; }

public string Bairro { get; internal set; }

public string Cidade { get; internal set; }

}

}

Voltaremos para o método Main() para popularmos as informações de endereço e associarmos com o cliente. Já instanciaremos o objeto de LojaContext para persistirmos.

static void Main(string[] args)

{

var fulano = new Cliente();

fulano.Nome = "Fulaninho de Tal";

fulano.EnderecoDeEntrega = new Endereco()

{

Numero = 12,

Logradouro = "Rua dos Inválidos",

Complemento = "sobrado",

Bairro = "Centro",

Cidade = "Cidade"

};

using (var contexto = new LojaContext())

{

contexto.Clientes.Add(fulano);

contexto.SaveChanges();

}

}

Dentro da classe LojaContext iremos criar a propriedade para o cliente.

namespace Alura.Loja.Testes.ConsoleApp

{

public class LojaContext : DbContext

{

public DbSet<Produto> Produtos { get; set; }

public DbSet<Compras> Compras { get; set; }

public DbSet<Promocao> Promocoes { get; set; }

public DbSet<Cliente> Clientes { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=LojaDB;Trusted\_Connection=true;")

}

}

}

Fizemos mais uma evolução da aplicação. Agora precisamos sincronizar com o banco de dados.

Quando o entity começa a mapear as classes a partir das suas propriedades, ele também navega por essas propriedades. No caso a classe Endereco envolvida no relacionamento não definiu sua chave primária.

Shadow properties são propriedades que não existem na classe que representa sua entidade, mas existem na tabela.

Correta!

Alternativa correta

Para criar a *shadow property* UltimoLogin do tipo DateTime na classe Usuario, colocamos o código abaixo no método OnModelCreating.

modelBuilder

.Entity<Usuario>()

.Property<DateTime>("UltimoLogin");

Isso aí! Lembre-se de definir o tipo da propriedade (no caso DateTime).

Nas aulas anteriores conversamos sobre relacionamentos. Nessa aula usaremos esses relacionamentos para montarmos *querys* um pouco mais complexas.

Como já fizemos em aulas anteriores, vamos mover todo o código contido no método Main() para um outro método chamado UmParaUm(). Em seguida criaremos o objeto de LojaContext e uma promoção para janeiro.

class Program

{

static void Main(string[] args)

{

using(var contexto = new LojaContext())

{

var promocao = new Promocao();

promocao.Descricao = "Queima Total 2017";

promocao.DataInicio = new DateTime(2017, 1, 1);

promocao.DataTermino = new DateTime(2017, 1, 31);

}

}

// ...

}

Adicionaremos produtos nessa promoção, mas ao invés de criarmos novos produtos, pegaremos do banco de dados todos os produtos da categoria "Bebidas". Estamos acostumados a pegar todos os produtos usando apenas o ToList(), mas o ***LINQ*** do **C#** possui métodos que nos auxiliam. Um desses métodos é o Where() que recebe uma expressão lambda **booleana**. Depois de pegar os produtos, adicionaremos dentro da promoção e a promoção no contexto. A classe ficará da seguinte maneira:

class Program

{

static void Main(string[] args)

{

using(var contexto = new LojaContext())

{

var promocao = new Promocao();

promocao.Descricao = "Queima Total Janeiro 2017";

promocao.DataInicio = new DateTime(2017, 1, 1);

promocao.DataTermino = new DateTime(2017, 1, 31);

var produtos = contexto

.Produtos

.Where(p => p.Categoria == "Bebidas")

.ToList();

foreach(var item in produtos)

{

promocao.IncluiProduto(item);

}

contexto.Promocoes.Add(promocao);

}

}

// ...

}

Após o contexto.Promocoes.Add(promocao) chamaremos o método ExibeEntries() para verificarmos como o Entity está tratando as informações.

class Program

{

static void Main(string[] args)

{

using(var contexto = new LojaContext())

{

var promocao = new Promocao();

promocao.Descricao = "Queima Total Janeiro 2017";

promocao.DataInicio = new DateTime(2017, 1, 1);

promocao.DataTermino= new DateTime(2017, 1, 31);

var produtos = contexto

.Produtos

.Where(p => p.Categoria == "Bebidas")

.ToList();

foreach(var item in produtos)

{

promocao.IncluiProduto(item);

}

contexto.Promocoes.Add(promocao);

ExibeEntries(contexto.ChangeTracker.Entries());

}

}

// ...

}

Veremos que ele buscou apenas produtos com a categoria "Bebidas" (caso não tenha nada no banco cadastre alguns produtos para testar). Em seguida ele adicionou em promocao e deixou com o estado ***Added***.

Como tudo parece estar funcionando, adicionaremos o contexto.SaveChanges(). Fora do escopo do contexto, criaremos outro objeto de LojaContext() para mostrarmos os produtos cadastrados na promocao. Quando fechamos um contexto do Entity, os objetos deixam de ser rastreados por aquele contexto.

class Program

{

static void Main(string[] args)

{

using(var contexto = new LojaContext())

{

var promocao = new Promocao();

promocao.Descricao = "Queima Total Janeiro 2017";

promocao.DataInicio = new DateTime(2017, 1, 1);

promocao.DataTermino = new DateTime(2017, 1, 31);

var produtos = contexto

.Produtos

.Where(p => p.Categoria == "Bebidas")

.ToList();

foreach(var item in produtos)

{

promocao.IncluiProduto(item);

}

contexto.Promocoes.Add(promocao);

ExibeEntries(contexto.ChangeTracker.Entries());

contexto.SaveChanges();

}

using(var contexto2 = new LojaContext())

{

var promocao = contexto2.Promocoes.FirstOrDefault();

Console.WriteLine("\nMotrando os produtos da promoção...");

foreach(var item in promocao.Produtos)

{

Console.WriteLine(item.Produto);

}

}

}

// ...

}

Após executarmos a aplicação veremos que o primeiro contexto fez o SELECT em Produtos com a categoria "Bebida", os produtos foram adicionados no ***Change Tracker***, e então foi feito o INSERTem Promocoes e em PromocaoProduto.

Já o segundo contexto fez o SELECT na primeira promoção e não mostrou nada. Olhando no banco de dados vemos que os produtos foram adicionados no ProdutoPromocao, lembrando que para isso é necessário que os produtos existam no banco.

O próximo passo é descobrir por que ele não está mostrando os produtos.

Queremos saber porque os produtos da promoção não estão sendo recuperados, quando buscamos com outro contexto do Entity. Na verdade, esse tipo de comportamento de não recuperar as entidade relacionadas junto com o SELECT é padrão em várias ferramentas de ORM.

Se as ferramentas de ORM não tivessem esse tipo de comportamento, seria buscado para a memória todas as entidades relacionadas a classe, o que poderia trazer problemas de performance. Temos que informar que queremos entidades relacionadas explicitamente por meio de métodos.

Antes começarmos a trabalhar na busca, extrairemos todo o código que insere a promoção no banco para um método chamado IncluirPromocao(). Deixaremos no Main() apenas o código onde buscamos pelos produtos da promoção:

class Program

{

static void Main(string[] args)

{

using(var contexto2 = new LojaContext())

{

var promocao = contexto2.Promocoes.FirstOrDefault();

Console.WriteLine("\nMotrando os produtos da promoção...");

foreach(var item in promocao.Produtos)

{

Console.WriteLine(item.Produto);

}

}

}

// ...

}

O método que precisamos invocar é o Include(), que incluirá a entidade relacionada. Porém como queremos os produtos, precisamos chamar a entidade PromocaoProduto. Para inclui-lá usaremos o método ThenInclude(), ou seja, buscaremos uma Promocao que busca uma PromocaoProduto, onde podemos acessar o Produto. A busca será feita da seguinte forma:

class Program

{

static void Main(string[] args)

{

using(var contexto2 = new LojaContext())

{

var promocao = contexto2

.Promocoes

.Include(p => p.Produtos)

.ThenInclude(pp => pp.Produto)

.FirstOrDefault();

Console.WriteLine("\nMotrando os produtos da promoção...");

foreach(var item in promocao.Produtos)

{

Console.WriteLine(item.Produto);

}

}

}

// ...

}

Executaremos a aplicação. Como resultado, veremos um SELECT em Promocoes, mas além disso, temos outro SELECT para trazer os produtos. É dessa forma que o Entity utiliza para buscar entidades relacionadas e manter a performance.

Veremos como fazer essas buscas com relacionamentos **um para um** e **um para muitos**.

O método Include possui uma segunda sobrecarga, que permite informarmos como argumento de entrada uma string com o nome da propriedade de navegação a ser incluída no *join*. A vantagem dessa abordagem é que não precisamos usar outros métodos ThenInclude para continuar a navegação em outras entidades. Por exemplo, para o exemplo Cliente x Conta, poderíamos fazer:

var lista = contexto.Clientes.Include("Contas.Conta");

A desvantagem é que se o nome da propriedade mudar, teremos que lembrar todos os lugares onde fizemos isso, porque não teremos ajuda do compilador.

Exploraremos o relacionamento **um para um** entre Cliente e Endereco. Como já fizemos anteriormente, vamos colocar todo o conteúdo do método Main() e um método chamado ExibeProdutosDaPromocao(). Dentro do Main() criaremos apenas um objeto de contexto do Entity e deixaremos os *Logger*.

static void Main(string[] args)

{

using (var contexto = new LojaContext())

{

var serviceProvider = contexto.GetInfrastructure<IServiceProvider>();

var loggerFactory = serviceProvider.GetService<ILoggerFactory>();

loggerFactory.AddProvider(SqlLoggerProvider.Create());

}

}

Buscaremos a propriedade Logradouro do endereço de um Cliente. Porém antes precisamos buscar o cliente, faremos isso usando o FirstOrDefault(). Como a propriedade EnderecoDeEntregajá está disponível, podemos acessar o Logradouro.

static void Main(string[] args)

{

using (var contexto = new LojaContext())

{

var serviceProvider = contexto.GetInfrastructure<IServiceProvider>();

var loggerFactory = serviceProvider.GetService<ILoggerFactory>();

loggerFactory.AddProvider(SqlLoggerProvider.Create());

var cliente = contexto.Clientes.FirstOrDefault();

Console.WriteLine($"Endereço de entrega: {cliente.EnderecoDeEntrega.Logradouro}");

}

}

Ao executarmos a aplicação ocorrerá um exceção. Isso aconteceu porque acessamos a propriedade Logradouro nula, o Entity não carregou essa propriedade. Incluiremos a entidade em nossa busca:

static void Main(string[] args)

{

using (var contexto = new LojaContext())

{

var serviceProvider = contexto.GetInfrastructure<IServiceProvider>();

var loggerFactory = serviceProvider.GetService<ILoggerFactory>();

loggerFactory.AddProvider(SqlLoggerProvider.Create());

var cliente = contexto

.Clientes

.Include(c => c.EnderecoDeEntrega)

.FirstOrDefault();

Console.WriteLine($"Endereço de entrega: {cliente.EnderecoDeEntrega.Logradouro}");

}

}

Executaremos a aplicação e veremos que funcionou perfeitamente. O SELECT que o Entity utilizou foi com LEFT JOIN, pelo fato do endereço não ser obrigatório para o cliente.

Continuaremos explorando as consultas com relacionamento, pegaremos as compras de um produto. Primeiro pegaremos um produto qualquer - no nosso caso o produto com uma compra era o com Id 9004 - passando o seu Id pelo Where().

static void Main(string[] args)

{

using (var contexto = new LojaContext())

{

var serviceProvider = contexto.GetInfrastructure<IServiceProvider>();

var loggerFactory = serviceProvider.GetService<ILoggerFactory>();

loggerFactory.AddProvider(SqlLoggerProvider.Create());

var cliente = contexto

.Clientes

.Include(c => c.EnderecoDeEntrega)

.FirstOrDefault();

Console.WriteLine($"Endereço de entrega: {cliente.EnderecoDeEntrega.Logradouro}");

var produto = contexto

.Produtos

.Where(p => p.Id == 9004)

.FirstOrDefault();

}

}

Com esse produto faremos um foreach na listagem produto.Compras. Porém não temos a lista de Compras pelo fato de que o Entity não nos obrigar a criá-la. Adicionaremos a lista na classe Produto:

namespace Alura.Loja.Testes.ConsoleApp

{

public class Produto

{

public int Id { get; internal set; }

public int Nome { get; internal set; }

public int Categoria { get; internal set; }

public int PrecoUnitario { get; internal set; }

public string Unidade { get; set; }

public IList<PromocaoProduto> Promocoes { get; set; }

public IList<Compra> Compras { get; set; }

public override string ToString()

{

return $"Produto: {this.Id}, {this.Nome}, {this.Categoria}, {this.PrecoUnitario}";

}

}

}

Antes de inciarmos a iteração pela lista, precisamos incluir as compras no Include().

static void Main(string[] args)

{

using (var contexto = new LojaContext())

{

var serviceProvider = contexto.GetInfrastructure<IServiceProvider>();

var loggerFactory = serviceProvider.GetService<ILoggerFactory>();

loggerFactory.AddProvider(SqlLoggerProvider.Create());

var cliente = contexto

.Clientes

.Include(c => c.EnderecoDeEntrega)

.FirstOrDefault();

Console.WriteLine($"Endereço de entrega: {cliente.EnderecoDeEntrega.Logradouro}");

var produto = contexto

.Produtos

.Include(p => p.Compras)

.Where(p => p.Id == 9004)

.FirstOrDefault();

Console.WriteLine($"Mostrando as compras do produto {produto.Nome}");

foreach(var item in produto.Compras)

{

Console.WriteLine(item);

}

}

}

Executaremos a aplicação e como resultado, veremos que o Entity buscou o produto com o Id **9004**, em seguida ele fez um INNER JOIN de Compras com Produtos para pegar as compras, e por último foi mostrado a compra. A única coisa que precisamos fazer é uma propriedade, onde navegaremos para o lado onde temos muitos.

Vimos como trabalhar com buscas de relacionamentos **muitos para muitos**, **um para um** e **um para muitos**. Mas e se quiséssemos buscar as compras acima de um valor determinado? É o nosso próximo passo!

**Queremos filtrar as compras acima de R$ 10,00. Não podemos usar o Where() para filtrar o preço da compra, justamente por estarmos acessando a partir de Produtos. O Include() também não será útil nessa situação. O que faremos é um segundo SELECT para filtrar as compras do produto que já buscamos.**

**Com o objeto contexto, chamaremos o método Entry() passando a referência de produto. Após, pegaremos a coleção da propriedade Compra com o método Collection(p => p.Compras). Em seguida faremos uma Query(), que filtrará com a condição Where(c => c.Preco > 10). Por último, carregaremos com Load() na referência passada no Entry(). O código ficará da seguinte maneira:**

**static void Main(string[] args)**

**{**

**using (var contexto = new LojaContext())**

**{**

**var serviceProvider = contexto.GetInfrastructure<IServiceProvider>();**

**var loggerFactory = serviceProvider.GetService<ILoggerFactory>();**

**loggerFactory.AddProvider(SqlLoggerProvider.Create());**

**var cliente = contexto**

**.Clientes**

**.Include(c => c.EnderecoDeEntrega)**

**.FirstOrDefault();**

**Console.WriteLine($"Endereço de entrega: {cliente.EnderecoDeEntrega.Logradouro}");**

**var produto = contexto**

**.Produtos**

**.Where(p => p.Id == 9004)**

**.FirstOrDefault();**

**contexto.Entry(produto)**

**.Collection(p => p.Compras)**

**.Query()**

**.Where(c => c.Preco > 10)**

**.Load();**

**Console.WriteLine($"Mostrando as compras do produto {produto.Nome}");**

**foreach(var item in produto.Compras)**

**{**

**Console.WriteLine("\t" + item);**

**}**

**}**

**}**

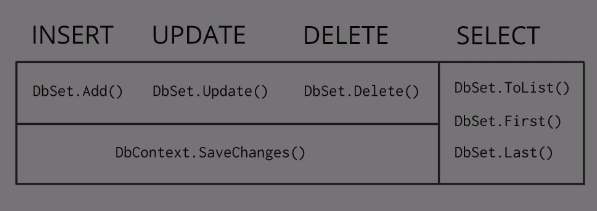
**Executaremos a aplicação com "Ctrl + F5". Poderemos ver que foi executado um SELECT em Produtos no banco de dados, buscando o produto com o Id passado no filtro Where(). Em seguida foi feito um segundo SELECT em Compras, filtrando pelo Id do produto passado no Entry() e pelo preço. Por fim, serão apresentados os produtos com preço acima de R$ 10,00.**

**Esse cenário é bastante utilizado quando queremos aplicar filtros em objetos relacionados da aplicação. Esta estratégia é conhecida como Carregamento Explícito, onde trazemos só o que nos interessa.**

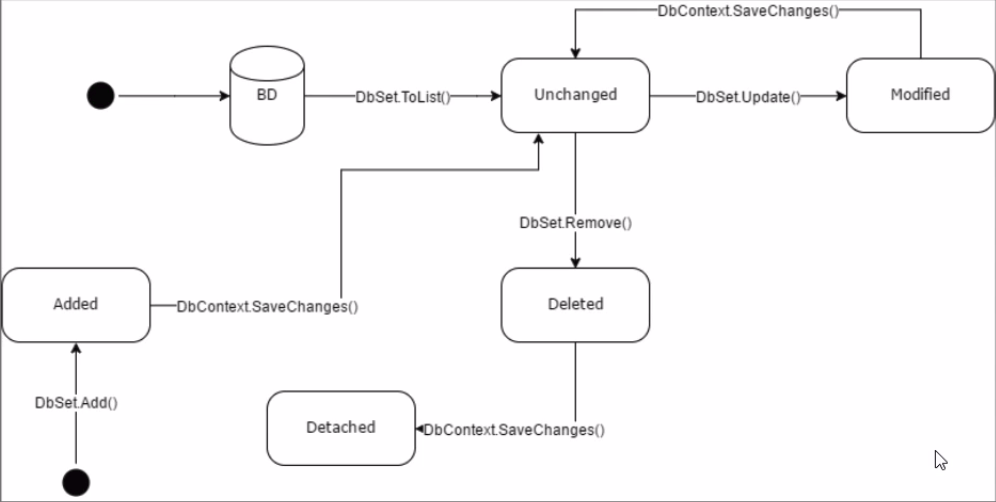
Finalizamos o curso de Entity Framework Core! Faremos um resumo sobre o que aprendemos durante o curso.

Vimos que para trabalhar com o modelo de negócios e persisti-lo no banco de dados era algo extremamente trabalhoso. O programador era responsável por gerenciar e manter o código que se comunicava com o banco. Qualquer mudança na lógica de negócio, gerava uma série de impactos nas classes de acesso aos dados.

Em seguida, Começamos o uso do Entity após a instalação dos pacotes, passando a substituir todo o trabalho, permitindo forcarmos na lógica de negócio. Aprendemos como o Entity gerencia os comandos SQL, trabalhando com a DML (INSERT, SELECT, UPDATE e DELETE) para manipular os dados.



Para gerenciar quais objetos e quais comandos SQL precisavam ser emitidos para o banco de dados, o Entity usava um recurso chamado ***Change Tracker***, onde era armazenado os estados de cada objeto.



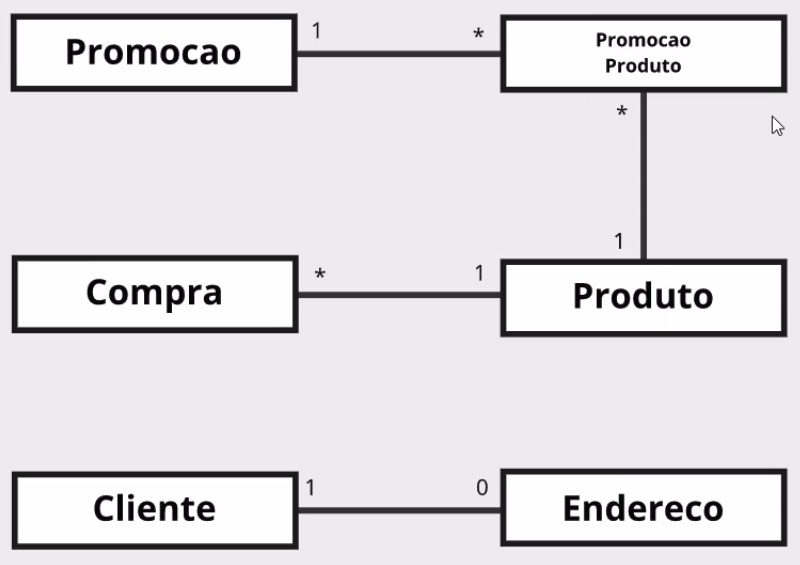
Enquanto a aplicação evoluía, era necessário sincronizá-la com o banco de dados, onde Entity também nos auxilia com a DDL (CREATE, DROP, ALTER, RENAME). Utilizamos o conceito de migrações (***Migrations***), onde as alterações no negócio são aplicadas ao banco. As migrações eram feitas por meio de classes, que tinha os métodos Up() para subir e Down() para descer de versão.

Instalamos no NuGet, um pacote para utilizarmos diversos comandos que trabalham com as migrações.

* Add-Migration
* Remove-Migration
* Update-Database

Todas as alterações no banco de dados eram armazenadas na tabela \_\_EFMigrationHistory. A tabela indicava quais migrações tinham sido aplicadas no banco de dados.

Após entender como o Entity funciona, aprendemos como ele trabalha e gerencia os relacionamentos entre classe. Os relacionamentos eram **um para um**, **um para muitos** e **muitos para muitos**. Porém o Entity Core não gerencia sozinho as tabelas JOIN, por isso contornamos criando uma classe específica.



Para recuperarmos dados relacionados, usamos métodos específicos:

* Include()
* ThenInclude()
* Load()

Esse foi o curso de Entity Framework Core.